DB Name	Query	Hit Count	Set Name
USPT	l1 and (table with look with aside$)	0	L11
USPT	l1 and (table with lookaside$)	0	L10
USPT	l1 and (tlb$)	1	L9 TLB
USPT	l1 and (pipeline$)	1	L8 – pipline
USPT	l1 and (interrupt$)	4	L7 - interupt
USPT	l1 and (table with lookup$)	0	L6
USPT	l1 and (table with lookup$ with interrupt$)	0	L5
USPT	l1 and (address$ with translat$)	3	L4 - addr tranl
USPT	l1 and (code with translat$)	1	L3 - code tras
USPT	l1 and (pipeline or interrupt$ or translat$ or (table adj lookup))	4	L2
USPT	(4812975 or 5678032 or 5751982 or 6006029).pn.	4	L1

WEST

Generate Collection

Search Results - Record(s) 1 through 4 of 4 returned.

□ 1. Document ID: US 6006029 A

L2: Entry 1 of 4 File: USPT Dec 21, 1999

DOCUMENT-IDENTIFIER: US 6006029 A
TITLE: Emulating disk drives of a first system on a second system

BSPR:
In the first approach, a layer of interpretive programs are interposed between the application programs and the operating system of the second system, that is, between the application programs and the execute level of the second operating system, The interpretive programs operate to translate each call, command or instruction of an application program into an operation or series of operations of the second operating system which are the equivalent of the operations of the first operating system that would have been performed in response to the same calls, commands or instructions from the application program.

DRPR:
FIG. 8 is the address translation mechanism and memory space mapping mechanism of the emulation mechanism.

DEPR:
The operations performed in First System 10 in execution of an Application Program (APP) 22 or a System Administrative Program (SAD) 24 are executed through a plurality of Tasks 30 and any program executing on First System 10 may spawn one or more Tasks 30. A Task 30 may be regarded as being analogous to a process, wherein a process is generally defined as a locus of control which moves through the programs and routines and data structures of a system to perform some specific operation or series of operations on behalf of a program. There is a Task Control Block (TCB) 32 associated with each Task 30 wherein the Task Control Block (TCB) 32 of a Task 30 is essentially a data structure containing information regarding and defining the state of execution of the associated Task 30. A Task Control Block (TCB) 32 may, for example, contain information regarding the state of execution of tasks or operations that the Task 30 has requested be performed and the information contained in a Task Control Block (TCB) 32 is available, for example, to the programs of Executive Program Tasks (EXP Tasks) 28 for use in managing the execution of the Task 30. Each Task 30 may also include an Interrupt Save Area (ISA) 34 which is used to store hardware parameters relevant to the Task 30.

DEPR:
5. Addresses and Address Translation

DEPR:
It will be noted, as described previously, that Software Active Queue (SAQ) 88, the Pseudo Device Queues (PSDQs) 86, and INTERPRETER 72 are provided to emulate the corresponding mechanisms of First System 10, that is, First System 10's input/output devices and central processing unit, as seen by Executive Program Tasks (EXP Tasks) 28 and Tasks 30. As such, Executive Program Tasks (EXP Tasks) 28 and Tasks 30 will provide memory addresses to the Pseudo Device Queues (PSDQs) 82 and INTERPRETER 72 according to the requirements of the native memory access and management mechanisms of First System 10 and will expect to receive memory addresses from Software Active Queue (SAQ) 88 and INTERPRETER 72 in the same form. Second System Kernel Processes (SKPs) 66, Lower Communications Facilities Layer Processes (LCFLPs) 78, the hardware elements of Second System 54 and other processes executing as native processes in Second System 54, however, operate

according to the memory addressing mechanisms native to Second System 54. As such, address translation is required when passing requests and returning requests between Emulator Executive Level (EEXL) 68 and Second System Kernel Level (SKernel) 64.

DEPR:
As described, INTEPRETER 70 is provided to interpret First System 10 instructions into functionally equivalent Second Second 54 instructions, or sequences of instructions, including instructions pertaining to memory operations. As such, the address translation mechanism is also associated with INTERPRETER 72, or is implemented as a part of INTERPRETER 72, and is indicated in FIG. 3 as Address Translation (ADDRXLT) 98 and will be described in detail in a following discussion.

DEPR:
The Queue Frames (QFs) 94 of Software Active Queue (SAQ) 88 and Pseudo Device Driver Queues (PSDQs) 86 differ in detail and the following will describe the Queue Frames (QFs) 94 of both, noting where the frames differ. Each Queue Frame (QF) 94 further includes a Task Control Block Pointer (TCBP) or Input/Output Request Block Pointer (IORBP) 38p, as previously described, a Priority Field (Priority) 108 containing a value indicating the relative priority of the interrupt or request. The Queue Frames (QFs) 94 of Software Active Queue (SAQ) 88 include a Flag Field (Flag) 108 containing a flag which distinguishes whether the Queue Frame (QF) 94 contains a Task Control Block (TCB) 32 or an Indirect Request Block (IRB) 36. Input/Output Request Blocks (IORBs) through their IRBs are generally given a higher priority than Task Control Blocks (TCBs). Exceptions may be made, however, for example, for clock and task inhibit Task Control Blocks (TCBs) 32, which must be given the highest priority.

DEPR:
As described above with reference to FIGS. 2 and 3, the First System 10 tasks and programs executing on Second System 54, Second System 54's native processes and mechanisms and the Second System 54 mechanisms emulating First System 10 mechanisms share and cooperatively use Second System 54's memory space in Second System Memory 58b. As a consequence, it is necessary for Second System 54, the First System 10 tasks and programs executing on Second System 54, and the emulation mechanisms to share memory use, management, and protection functions in a manner that is compatible with both Second System 54's normal memory operations and with First System 10's emulated memory operations. The emulation of First System 10 memory operations in Second System 54 in turn requires emulation of First System 10's memory management unit, that is, First System 10's hardware and software elements involved in memory space allocation, virtual to physical address translation, and memory protection in Second System 54. As described below, this emulation is implemented through use of Second System 52's native memory management unit to avoid the performance penalties incurred through a complete software emulation of First System 10's memory management unit.

DEPR:
As is well known, most systems operate upon the basis of virtual addresses and perform virtual to physical address translations relative to a predetermined base address, that is, by adding a virtual address as an offset address to the base address to determine the corresponding address in physical address space of the system. While First System 10 and Second System 52 may both use such addressing schemes, the actual addressing mechanisms of the two system may differ substantially, as may the memory protection schemes.

DEPR:
Referring to FIG. 8, and to FIGS. 2, 3, 5 and 6, therein is illustrated the mechanisms implemented on Second System 54 to emulate the memory access, protection, and management mechanisms of First System 10. It must be recognized in the following that the emulation of First System 10 memory operations on Second System 54 involves two differerent address conversion operations, one being the conversion of First System Virtual Addresses (FSVAs) 126 done by INTERPRETER 72 and the second being the conversion of First System Virtual Addresses (FSVAs) 126 done by Pseudo Device Drivers (PSDDs) 74. Each of these conversions is accomplished through translation and through mapping of First System 10 system and user memory areas into Second System 54 segments. The following will first describe the address translation operation performed by INTERPRETER 72, and then will describe the address translation operation

performed by Pseudo Device Drivers (PSDDs) 74.

DEPR:
First considering the process of INTERPRETER 72 address translation, as has been described above, each First System Virtual Address (FSVA) 126 is comprised of a Most Significant Bits field (MSB) 128 and an Address field (ADDR) 130 wherein Most Sigificant Bits field (MSB) 128 contains a bit field whose value identifies whether the address is directed to an executive memory area, that is, System Memory (SYSMEM) 110 area, or to an Independent-Memory Pool (IPOOL) 112. For example, the Most Sigificant Bits field (MSB) 128 may contain the value 0000 (0) when the request is directed to the System Memory (SYSMEM) 110 area and the value 0001 (1) when the request is directed to an Independent-Memory Pool (IPOOL) 112 area.

DEPR:
As indicated in FIG. 8, the First System Virtual Address (FSVA) 126 of a request which includes a memory access is provided to Address Translation (ADDRXLT) 98. Address Translation (ADDRXLT) 98 includes a Word To Byte Shifter (WBS) 148 which performs an initial translation of the First System Virtual Address (FSVA) 126 from the First System 10 format, in which addresses are on a per word basis, to a Second System 54 virtual address, in which addresses are on a per byte basis. This translation is performed by a left shift of the First System Virtual Address (FSVA) 126 and, in the translation and as indicated in FIG. 7, the value in the Most Sigificant Bits field (MSB) 128 field of the First System Virtual Address (FSVA) 126 is transformed from 0000 (0) or 0001 (1) to 0000 (0) or 0010 (2), respectively.

DEPR:
Having performed the translation of a First System Virtual Address (FSVA) 126 into a per byte address, Address Translation (ADDRXLT) 98's Ring Adder (RNGA) 150 will read a System Status Register (SSR) 152 which, among other information, contains a Ring Number (RNG) 154 which contains a value indicating the First System 10 ring in which the task is executing, that is, a value of 0, 1, 2 or 3. As described, Ring 0 is reverved for system operations while Rings 1, 2 and 3 are used for user tasks. If the task is executing in Ring 0, that is, in system space, Ring Adder (RNGA) 150 will add 3 to the value (0 or 2) contained in Most Significant Bits field (MSB) 128 of the shifted First System Virtual Address (FSVA) 126. If the task is not executing in Ring 0, that is, is executing in Rings 1, 2, or 3 and thus in user task space, Ring Adder (RNGA) 150 will add 4 to the value (0 or 2) contained in Most Significant Bits field (MSB) 128 of the shifted First System Virtual Address (FSVA) 126. The final result will be a byte oriented First System Virtual Address (FSVA) 126 having a Most Significant Bits field (MSB) 128 which contains a value of 3, 4, 5 or 6, thereby indicating the Second System 54 memory space segment in which the address lies and an Address (ADDR) field 130 identifying a location within the segment.

DEPR:
The INTERPRETER 72 translation process always generates adddresses in segments 5 and 6 for user task addresses, but because of dynamic detaching and attaching of Independent Memory Pools (IPOOLs) 112, the same addresses will refer to different Independent Memory Pools (IPOOLs) 112. The mapping of system memory areas remains the same, however, when switching from Task Control Block (TCB) 32 to Task Control Block (TCB) 32, so that the INTERPRETER 72 generated addresses in segments 3 and 4 always refer to the same locations.

DEPR:
As has been described, each Pseudo Device Driver Queue (PSDQ) 86 is associated with a corresponding Second System Kernel Process (SKP) 66 which executes the requests in the Pseudo Device Driver Queue (PSDQ) 86 and any Pseudo Device Driver Queue (PSDQ) 86 may contain requests from a plurality of tasks, each task in turn being associated with and executed in an Independent-Memory Pool (IPOOL) 112 area which is mapped into a Second System 54 memory segment by address translator (ADDRXLP) 96 which includes a Server Pool Descriptor Linked Set (SPDLS) associated with the Pseudo Device Driver Queue (PSDQ) 86, Task Control Block (TCB) 32, Segment Descriptor Table 156, and Memory Pool Array 162.

DEPR:
It may be seen from the above descriptions, therefore, that, for any first system virtual address generated by a First System 10 task executing on Second System

54, INTERPRETER 72 will translate the First System 10 virtual address into a byte oriented virtual address containing a virtual address location within a segment and identifying a Segment 3, 4, 5 or 6 containing the location. The INTERPRETER 72 mapping of segments via ADDRXLT98 will in turn map each segment identified by an address translation into an Independent Memory Pool Identification (IPOOLID) 160 for the current task. The Segment/Independent Memory Pool mapping mechanism (i.e., ADDRXLP96) of the Pseudo Device Driver (PSDD) 74 executing the task request associated with the First System 10 virtual address will map the segment identified by the address translation mechanism to a current Independent Memory Pool (IPOOL) 112 location in System 54's memory by providing the base address corresponding to the Independent Memory Pool Identification (IPOOLID) 160.

DEPR:
The structure and operation of the present invention are further described by reference to the following Appendices which contain program listings contained in Appendices A, B, C and D published in U.S. Pat. No. 5,619,682 issued on Apr. 8, 1997 for EXECUTING NETWORK LAYERED COMMUNICATIONS OF A FIRST SYSTEM ON A SECOND SYSTEM USING A COMMUNICATION BRIDGE TRANSPARENT TO THE DIFFERENT COMMUNICATION LAYERS, which are hereby incorporate herein by reference and all right, title and interest to which is assigned to Bull HN Information Systems Incorporated, of Billerica, Mass. The Appendices in U.S. Pat. No. 5,619,682 for Memory Queue Interface (MQI) 84 and Escape/Call Mechanism (EscapeC) 100, Pseudo Network Layer (PNL) 76a residing and executing in First System Executive Level (FEXL) 16 as a native First System 10 program module and Pseudo Network Driver (PND) 76b, INTERPRETER 72 and the address/segment translation and mapping functions.

Full | Title | Citation | Front | Review | Classification | Date | Reference | Claims | KWIC | Draw Desc | Image

□ 2. Document ID: US 5751982 A

L2: Entry 2 of 4 File: USPT May 12, 1998

DOCUMENT-IDENTIFIER: US 5751982 A
TITLE: Software emulation system with dynamic translation of emulated instructions for increased processing speed

ABPL:
The execution time overhead of software emulation is reduced by selecting frequently emulated instruction sequences in the software being emulated, and translating those instruction sequences into the instruction set of the emulating processor. In a first phase, frequently repeated sequences of emulated computer code are identified and selected for translation. In a second phase, the selected sequences from the instruction set of the emulated processor are translated into equivalent sequences for the instruction set of the emulating processor. In a third phase, the instruction sequence of the emulating processor is executed in lieu of emulating the original instructions from the emulated software.

BSPR:
The present invention is directed to emulation systems which execute software instructions designed for a specific instruction set on a processor which supports a different instruction set, and more particularly to the use of dynamic translation of instructions to increase emulation performance.

BSPR:
In accordance with the present invention, the processing overhead of an emulating system is reduced by selecting frequently emulated instruction sequences in the software being emulated and translating those instruction sequences into the native instruction set of the emulating processor. The dynamic translation process of the present invention is implemented in three main phases. In the first phase, an identification is made of those code sequences in the emulated software that are frequently repeated. This can be done, for example, by recording all program counter values that are produced by instructions that cause a non-sequential change in the value of the program counter for the emulated

software. On a periodic basis, the recorded program counter values are analyzed, to identify code sequences which are emulated frequently enough to warrant dynamic translation. The program counter values which are selected identify the starting point of emulated code sequences to be translated.

BSPR:
In the second phase, the selected code sequences are translated from the instruction set of the emulated processor into the instruction set of the emulating processor. For each emulated instruction in a selected code sequence, its equivalent code sequence in the native instruction set is obtained from the emulator's set of semantic routines, by indexing into the dispatch table with a binary code for the emulated instruction. The successively retrieved code sequences are cumulatively stored in an instruction buffer, until each instruction in the selected sequence has been translated.

BSPR:
In the final phase, the translated sequence is executed in place of emulating the untranslated sequence. This is done by assigning each newly translated code sequence an available operation code from the emulated instruction set. The start of the untranslated instruction sequence is replaced with the value of this operation code, and the address of the translated instruction sequence is inserted in the dispatch table as the corresponding entry for this operation code. From that point on, the emulation software automatically executes the translated code sequence in lieu of emulating the untranslated code sequence, resulting in a substantial improvement in the time required to perform the emulation.

BSPR:
As a further feature of the invention, the computation of gratuitous values that are not employed during execution, such as flag values for emulated condition code registers, can be eliminated to further optimize the translated code sequence. In the implementation of this feature, an analysis of the emulated code sequence being translated is performed to identify any emulated instructions in the sequence which unnecessarily modify emulated condition flags. The modification of an emulated condition flag is unnecessary when a subsequent emulated instruction in the sequence modifies the same flag and no intervening emulated instruction either refers to the flag value or causes a non-sequential change to the emulated program counter value. For each emulated instruction where these conditions are met, an alternative native code sequence, which does not include the instructions to compute the unnecessary condition flag values, is employed in the translation of that emulated instruction.

DRPR:
FIG. 2 is a block diagram similar to FIG. 1, which depicts the conventional translation of a code block;

DRPR:
FIG. 3 is another block diagram similar to FIG. 1, depicting dynamic translation in accordance with the present invention; and

DRPR:
FIG. 4 is a block diagram depicting the execution of a translated code block.

DEPR:
Referring to FIG. 2, an example of a code block is represented by the shaded area 22. In a conventional emulation process, the instructions in a code block are individually translated each time they are encountered during the emulation process. As each instruction is fetched by the dispatcher 14, it functions as an address to the dispatch table 16. The dispatcher 14 retrieves the pointer stored at the addressed portion of the table 16, and calls the corresponding semantic routine to which the pointer refers, for execution. In the specific example of FIG. 2, the code block 22 consists of three instructions. As is conventional in most programs, each instruction is represented by a unique numerical value, known as its operation code, or opcode. For this example, the three instructions have the opcodes 2030, 2031 and 4e75. These three instructions respectively result in three dispatch operations to separately retrieve and execute three semantic routines for those opcodes, respectively. This need to continually dispatch each instruction as it is issued by the application program presents a significant amount of processing overhead in the overall operation of the emulator.

DEPR:
In accordance with the present invention, this processing overhead can be considerably reduced through dynamic translation of selected code blocks in the emulated application program. To implement this feature, the code blocks which are emulated frequently enough to warrant dynamic translation are first identified. This can be carried out by recording program counter values that produce non-sequential changes in the sequence of instructions being emulated. Whenever an instruction from the CISC code is emulated that results in a non-sequential change to the program counter, the new program counter value is recorded, for example by pushing its value onto a programmatic stack. In essence, each recorded program counter value represents the starting point of a new code block. Whenever there is a break in the operation of the emulator, for example as the processor stops emulation to service a special event or an interruption, the accumulated values are removed from the stack and analyzed to identify code blocks that are emulated more than a defined number of times within a predetermined time window. For example, if a particular block is emulated more than 256 times within a period of about 16 milliseconds, it may be selected. Any suitable approach can be employed to select the program counter values that identify the code blocks which occur with sufficient frequency. In the preferred embodiment of the invention, as each recorded program counter value is removed from the stack, its value is used as a hash index into a table of frequency counts, and the corresponding entry in the table is incremented by one. When the count is incremented beyond a predetermined threshold value, the program counter value corresponding to that table entry is placed on a list of code blocks to be dynamically translated.

DEPR:
The selected values identify the starting points of emulated code sequences, or code blocks, that are to be dynamically translated. Each code block begins with an instruction that is the target of a non-sequential change in the value of the program counter, and ends with the last instruction that can be reached by any code path that is wholly contained within the code block. Once a code block has been selected for dynamic translation, each instruction in the selected code block is translated from the instruction set of the emulated processor into the native code. For each instruction in the code block, its equivalent semantic routine is identified in the native code, using the dispatch table 16. Referring to FIG. 3, as the first instruction in the code block, in this example 2030, is translated, its corresponding semantic routine 2030 is stored in a buffer 26. As the next instruction in the code block is translated, its semantic routine 2031 is appended to the instructions stored in the buffer 26. This process is repeated until each instruction in the selected code block has been translated and its corresponding semantic routine stored in the instruction buffer 26.

DEPR:
This entire dynamic translation procedure, namely (a) the analysis of program counter values, (b) the identification of frequently emulated code blocks, and (c) the translation and storage of selected code blocks in the buffer, preferably takes place during the interruption of the emulation, i.e. prior to the time that the event which interrupted the emulation is serviced.

DEPR:
Once the entire code block has been translated in this manner, the translated sequence is executed in place of emulating the individual instructions of the untranslated sequence. To do so, the newly translated code sequence is assigned an opcode from the instruction set of the emulated processor. For example, the instruction set for the Motorola 68000 series of microprocessors employs a 16-bit opcode. This presents the capability for over 65,536 different possible opcodes that can be represented. Of this number, about 50,000 are actually employed for the instruction set, leaving a remainder of about 15,000 unassigned operation codes that are available for other uses. In the implementation of the present invention, each newly translated code sequence that is stored in an instruction buffer 26 is assigned one of the available operation codes from the emulated instruction set. The beginning instruction of the untranslated instruction sequence 22 is overwritten with the value of this new, unassigned operation code. Referring to the example of FIG. 4, the opcode 6001 is assumed to be one such unassigned, and hence available, opcode. It is substituted for the opcode 2030 of the first instruction in the code block 22. In the dispatch table 16, the address of the translated instruction sequence in the buffer 26 is inserted as the

corresponding entry for the new operation code. Subsequently, whenever the application program issues the first instruction in the translated code block, the dispatcher 14 will point to the translated code sequence, which will be automatically executed by the emulation software, in lieu of emulating the untranslated code sequence. Consequently, only a single call to the dispatcher is made, rather than a separate call for each instruction in the code block, resulting in a substantial improvement in the time required to perform the emulation.

DEPR:
As a feature of the present invention, the performance of the emulator can be further improved by eliminating the computation of gratuitous condition code values under those conditions in which their elimination results in semantically identical results in the emulation of the code block. This feature of the invention can be implemented during the second phase of the dynamic translation process, i.e. while the instructions of the selected code block are being translated. During this phase, the instructions in the emulated code are analyzed to identify those situations in which a subsequent instruction, or set of instructions, modifies all of the condition code flags that are set by the instruction under analysis. If no intervening instructions refer to the condition code flags, or cause a potential non-sequential change in the program counter value, the computation of the condition code flags can be eliminated. In this situation, an alternative semantic routine, which does not include computation of the condition code flag values, is employed for the translation of the instruction under analysis. The alternate semantic routine for an emulated instruction can be stored immediately following the primary semantic routine for that instruction. In the case of some CISC instructions, the elimination of condition code flag computations can reduce the number of native instructions in the semantic routine by two-thirds, resulting in appreciable speed and code size savings.

DEPR:
From the foregoing, it can be seen that the present invention employs dynamic translation to significantly improve the execution time of a software emulator. The dynamic translation results in a single dispatch operation for an entire code block of the emulated code, thereby significantly reducing the dispatch overhead associated with the emulation process. In the implementation of the invention, it is desirable to translate as many instructions as possible in a code block. The longer the sequence of instructions that are dynamically translated, the greater the increase in the emulation speed. This increased performance is due to the elimination of instruction decoding overhead, as well as increased locality within the target processor's instruction caches. More particularly, the sequence of native instructions that comprises a translated code block will generally fit within the available instruction cache memory of a RISC processor. When the translated block contains frequently iterated loops of code, a large stream of instructions will be executed directly out of cache memory at a much greater overall instruction throughput. Conversely, if that same block were to be emulated, there will likely be cache conflicts generated between the emulator's dispatcher, the dispatch table, and the semantic routines, and these delays will be multiplied by the number of loop iterations within the block.

CLPR:
2. The method of claim 1 further including the steps of analyzing the instructions in said sequence to determine whether a given instruction calculates values that are recalculated by a subsequent instruction in said sequence and which are not utilized prior to said recalculation, and translating said given instruction into a corresponding instruction in said first set which does not calculate said values.

CLPR:
5. The method of claim 4 wherein said counting step comprises the steps of loading a value associated with said non-sequential change in a stack upon each detected occurrence, emptying said stack in response to an interruption in an emulation operation, and counting the number of occurrences of each value in said stack.

CLPR:
7. The system of claim 6 further including means for analyzing the instructions in said sequence to determine whether a given instruction calculates values that

are recalculated by a subsequent instruction in said sequence and which are not utilized prior to said recalculation, and means for translating said given instruction into a corresponding instruction in said first set which does not calculate said values.

CLPV:
translating each of the instructions in said sequence into one or more corresponding instructions in said first set of instructions;

CLPV:
cumulatively storing the corresponding instructions from said translated instructions as a group; and

CLPV:
translating each of the instructions in said sequence into one or more corresponding instructions in said first set of instructions;

CLPV:
cumulatively storing the corresponding instructions from said translated instructions as a group; and

CLPV:
a dispatcher for translating instructions issued from said first set of instructions into one or more corresponding instructions in said second set of instructions;

CLPV:
a buffer which cumulatively stores instructions from said translated instructions that respectively correspond to the instructions in an identified sequence; and

CLPV:
a dispatcher for translating instructions issued from said first set of instructions into one or more corresponding instructions in said second set of instructions;

CLPV:
a buffer which cumulatively stores instructions from said translated instructions that respectively correspond to the instructions in an identified sequence; and

Full | Title | Citation | Front | Review | Classification | Date | Reference | Claims | KWIC | Draw Desc | Image

□ 3. Document ID: US 5678032 A

L2: Entry 3 of 4 File: USPT Oct 14, 1997

DOCUMENT-IDENTIFIER: US 5678032 A
TITLE: Method of optimizing the execution of program instuctions by an emulator using a plurality of execution units

BSPR:
As is well known in the art, there are a number of different techniques which are used to enable applications to be executed more efficiently, such as applications written for a specific computer architecture to be emulated on a host system having a different architecture. In the case of emulation applications, these techniques include static recompilation, dynamic recompilation and interpretive emulation. In the latter, an emulator is written usually to run on the host computer which translates a sequence of emulated program instructions intended to be executed on the emulated architecture into one or more instructions in the host's instruction language to perform the same function.

DEPR:
In the emulated system, each task 30 utilizes a plurality of data control structures, such as a task control block (TCB) structure 32, an indirect request

block (IRB) structure 36, an input/output request block (IORB) structure 38 and a resource control table (RCT) structure 40. The task control block (TCB) structure 32 contains information pertaining to the state of execution of the associated task as web as pointers to interrupt save areas for storing hardware parameters related to the task. The indirect request block (IRB) structure 36 contains information defining the operation requested by an associated task and includes pointers identifying the task and its associated task control block (TCB) and a pointer to the associated IORB structure.

DEPR:
As indicated in FIG. 1b, the next layer within the user level is the emulator executive level 68. This level includes certain components present in the emulated system which have been transformed into new mechanisms which appear to the remaining unchanged components to operate as the original unchanged components of the emulated system. At the same time, these new mechanisms appear to the components of the kernel level 64 as native components with which the host system is accustomed to operate, As shown, the components include the interpreter 72, an emulator monitor call unit (EMCU) 73, an emulator interrupt handler unit (EEHU) 74, a plurality of servers 90 through 98, and a plurality of pseudo device drivers (PSDD) 86 arranged as shown.

DEPR:
The kernel process manager component 70 includes an exception/interrupt services mechanism. The mechanism handles both interrupts and exception conditions (i.e. unexpected events caused by a process such as illegally addressing memory which can cause a segment violation signal). Exceptions can happen "in the middle" of the execution of a RISC instruction and the host system normally attempts to restart the instruction after handling the exception condition.

DETL:

APPENDIX I BLOCK F # Start of Instruction # (normal start, bypassed in optimized code) soi: sli. dt,iw,2 # iw * 4 lx w0,dt,top, # fetch XA entry soi.sub.-- p2: mtlr w0 # prepare to splatter bge 0,soi.sub.-- 0 # branch for opcodes 0-0x7ff (executed by ICU) rlinm w2,iw,28,21,28 # get index to XE table soi.sub.-- 1: P1 lux wo,w2,xep # fetch XE pointer (MAS) (f) Q1 lha dt,2(p) # fetch @ (P + 2) (next instr. if current instr. is one word) (b) P2 l w1,4(w2) # fetch XE pointer (RAS) (f) Q2 lha z,4(p) # fetch @ (P + 4) (next inst. if current inst. is two words) (b) (start of instruction, optimized code) soi.sub.-- 1b: P3 mtctr w0 # move memory AS XE address to CTR (f) Q3 sli. dt,dt,2 # [@(P + 2)]*4 (b) P4 st p,ph # save copy of P for traps (f) Q4 sli z,z,2 # [@(P + 4)] * 4 (b) bbfr 12 # jump to entry pt if no interrupt (f) (go to XA fragment/BLOCK D) BLOCK D # Base register effective address calculation, operand fetch P5 mr ea,bl # centralize b register (f) Q5 lx w0,top,dt # fetch next IW XA (b) P6 mr z,dt # centralize next IW (b) bbtl 15,.sub.-- xvba.sub.-- vp # if v pool, go to addr convert routine (f) Q6 lha dt,0(ea) # fetch word operand (f) P7 cal p,2(p) # advance P to next IW (b) bctr # splatter to XE (f) (continue in XE fragment/BLOCK E) BLOCK E # Load R register execution routine P8 srai iw,z,2 # set next IW (b) Q8 mtlr w0 # prepare to splatter (b) P9 mr rl,dt # move operand to register (f) Q9 rlinm w2,iw,28,21,28 # get index to XE table (f) bge 0,soi.sub.-- 0 # branch for opcodes 0-0x7fff (f) P10 lha dt,2(p) # fetch IW + 2 (b) Q10 lux w0,w2,xep # fetch XE pointer (MAS) (f) P11 lha z,4(p) # fetch IW + 4 (b) Q11 l wl,4(w2) # fetch XE pointer (RAS) (f) b soi.sub.-- 1b # return to fetch routine (f) (return to start of instruction/BLOCK F, label APPENDIX II BLOCK F # # Start of Instruction # (normal start, bypassed in optimized code) soi: sli. dt,iw,2 # iw * 4 lx w0,dt,top # fetch XA entry soi.sub.-- p2: mtlr w0 # prepare to splatter bge 0,soi.sub.-- 0 # branch for opcodes 0-0x7fff rlinm w2,iw,28,21,28 # get index to XE table (start of instruction, optimized code) soi.sub.-- 1: P1 lux w0,w2,xep # fetch XE ptr (MAS) (f) Q1 lha dt,2(p) # fetch @(P + 2) (b) P2 l w1,4(w2) # fetch XE ptr (RAS) (f) Q2 lha z,4(p) # fetch @(P + 4) (b) soi.sub.-- 1b: P3 mtctr w0 # move memory AS XE addr to CTR (f) Q3 sli. dt,dt,2 # [@(P + 2)] * 4 (b) P4 st p,ph # save copy of P for traps (f) Q4 sli z,z,2 # [@(P + 4)] * 4 (b) bbfr 12 # jump to entry point if no intr (f) (go to XA fragment/BLOCK D) BLOCK D # Base register + index register # effective address calculation, operand fetch P5 cax ea,b1,r2 # add index to base (f) Q5 mr z,dt # centralize next IW (b) P6 lx w0,top,z # fetch next IW XA (b) bbtl 15,.sub.-- xvba.sub.-- vp # if v pool, go to addr convert routine (f) Q6 lha dt,0(ea) # fetch word operand (f) Segment Violation occurs here, continue in Signal Catcher C code Segment Violation Signal Catcher C Code (partial) /* handle rare case of word prefetch at last byte of AIX segment */ /* case 1: I pool, multiple sources of segvio */ /* test lower 12 bits of register ea for all 1 (last byte of page) */ if((scp->sc.sub.-- jmpbuf.jmp.sub.-- context.gpr[25] & 0x0fff) = 0xfff) { /* test instruction is : `lha dt,0(ea)`*/ longbase = (long *) scp->sc.sub.-- jmpbuf.jmp.sub.-- context.iar; if (*longbase = 0xab190000) { /* set link register to followang instruction */ scp->sc.sub.-- jmpbuf.jmp.sub.-- context.lr = scp->sc.sub.-- jmpbuf.jmp.sub.-- context.iar+4; /* direct interpreter to fixup code */ scp->sc.sub.-- jmpbuf.jmp.sub.-- context.iar=(long)fetch8; return; } } /* case 2: V pool, single instance */ if(scp->sc.sub.-- jmpbuf.jmp.sub.-- context.iar = (long) xva16.sub.-- segvio) { scp->sc.sub.-- jmpbuf.jmp.sub.-- context.lr=scp->sc.sub.-- jmpbuf.jmp.sub. -- context.iar+4; scp->sc.sub.-- jmpbuf.jmp.sub.-- context.iar=(long)fetch8v; return; } Assembly code in interpreter unit to refetch byte operand # This code handles extraneous segvio's caused by the # sequence of a byte operand prefetch performed # via a 16 bit read on the very last byte of an AIX # segment. It is continuation code that the interpreter # is directed to by the segvio signal catcher when the # sequence is detected. # .globl fetch8v .globl fetch8 fetch8v: cax ea,ea,dt # add M:seg.sub.-- base to addr fetch8: lbz dt,0(ea) # fetch byte operand sli dt,dt,8 # move to right middle byte exts dt,dt # sign extend br # return to original sequence BLOCK D -(cont'd) Return here after Signal Catcher/fetch 8 code recovery, or continue from BLOCK D (Q7) cal p,2(p) # advance P to next IW (b) bctr # splatter to XE (f) (continue in XE fragment/BLOCK E) BLOCK E # Load R register halfword execution routine (P8) srai iw,z,2 # set next IW (b) (Q8) mtlr w0 # prepare to splatter (b) (P9) srai rl,dt,8 # store left byte (f) (Q9) rlinm w2,iw,28,21,28 # get index to XE table (f) bge 0,soi.sub.-- 0 # branch for opcodes 0-0x7ff (f) (P10) lha dt,2(p) # fetch IW + 2 (b) (Q10) lux w0,w2,xep # fetch XE ptr (MAS) (f) (P11) lha z,4(p) # fetch IW + 4 (b) (Q11) l x1,4(w2) # fetch XE ptr (RAS) (f) b soi.sub.-- lb # return to fetch routine (f) (return to start of instruction/BLOCK F, label)

| Full | Title | Citation | Front | Review | Classification | Date | Reference | Claims | KWIC | Draw Desc | Image |

□ 4. Document ID: US 4812975 A

L2: Entry 4 of 4 File: USPT Mar 14, 1989

DOCUMENT-IDENTIFIER: US 4812975 A
TITLE: Emulation method

ABPL:
A method for emulating programs in a system includes a plurality of first and second data processors having different instruction word sets. An instruction which interrupts the operating system on the first data processor is defined. When the instruction is detected in a program running on the first data processor, it is determined whether or not the instruction is an instruction associated with an input/output macro instruction. If it is found, as a result of the determination, that this is the case, an interrupt is caused in a program running on the second data processor which controls the emulation, and the input/output macro instruction output from an emulated program is translated into an input/output macro instuction for the operating system, thereby implementing an emulation with a minimized overhead.

BSPR:
The present invention is materialized in a system comprising data processors, for example, a first data processor and a second data processor which have a different instruction word set, respectively. It is assumed that the user programs in the first data processor are processed under control of a first operating system and that those in the second data processor are processed under control of a second operating system. The programs in the first data processor are assigned as the programs to be emulated (to be referred to as emulated programs hereafter) and the specific instructions are defined in these emulated programs. Such an instruction like a supervisor call instruction, is one that causes an interrupt in the first operating system. When.. the interrupt

instruction is detected, it is examined to determine whether or not this is an interrupt instruction associated with an input/output control macro instruction. If it is found, as the result of the examination, that this is the case, an interrupt occurs in a program on the second data processor which controls the emulation. Then, the input/output control macro instructions issued by the emulated program are translated into the corresponding input/output control macro instructions for the second operating system.

BSPR:
Since the emulation can be conducted at the translated macro instruction level rather than at the machine language level, the input/output control instructions which occur frequently are executed with a relatively low overhead.

DEPR:
FIG. 1 depicts the operating system configuration. In this system, an operating system 10 of the native machine (new data processor) controls a plurality of native machine user programs 11-13 by use of an instruction word set. Furthermore, it also controls the jobs of, the target machine (old data processor). These operations are collectively referred to as an emulation job 20. In the emulation job 20, an emulated program 22 contains a target machine user program 24 and a target machine operating system 23 which controls the user program 24. The instruction word set to be used by the user program 24 and the operating system 23 is different from that used on the native machines described above. An emulation control program (to be abbreviated as ECP hereafter) 21 resides between the emulated program 22 and the native machine operating system 10 and functions as an interface program. The ECP 21 initiates the emulation by issuing an Execute Local (EXL) instruction to the emulated program 22 and translates the input/output control macro instructions issued in the target machine user program 24 into the corresponding input/output control macro instructions for the native machine operating system 10.

DEPR:
The EXL instruction format consists of an instruction code part (EXL), a B.sub.2 part comprising four bits for indicating a general-purpose register to be assigned as the base register, and a D.sub.2 part comprising 12 bits which indicate a binary value. When an EXL instruction is issued, the second operand address is register specified by the B.sub.2 part to the D.sub.2 part. The second operand address indicates an address on the main storage device at which a local execution list 41 (FIG. 4) is stored. The local execution list 41 comprises an area 411 for saving the emulation mode PSW necessary for the emulator operation, an area 412 for storing therein the new PSW for the ECP interrupt, a general-purpose register save area 413, an area for storing therein the first address of the jump table for the superviser call (SVC) instruction, and other areas. When the EXL instruction is executed, the pertinent values are set to the PSW, general-purpose register, etc. in the local execution list 41, then the system enters the local execution (emulator operation) mode.

DEPR:
In accordance with the emulation method of the present invention, since an input/output control instruction specified by the user program 24 in the emulated program 22 is issued in the form of an input/output macro instruction of the operating system 23 in the emulated program 22, and it is ordinarily an interrupt instruction, such as an SVC instruction, to the operating system 23; the input/output instruction is emulated at the macro instruction level rather than at the machine language level which has been adopted in the conventional emulation method.

DEPR:
This facilitates the translation operation for translating an input/output control macro instruction issued in the target machine user program 24 into the corresponding input/output control instruction for the native machine operating system 10; moreover, the necessity to perform the processing for each of a plurality of input/output instructions is obviated; thereby realizing an emulation with a minimized overhead.

DEPR:
In accordance with the present invention, when an SVC instruction is detected during the execution of the emulated program 22, the micro program for executing the emulation searches the entries of the local execution list 41 (FIG. 4) for

the SVC jump table first address entry 414. Each entry of the table 42 is referenced by use of the entry 414 stored in the local execution list 41. Then, the instruction is examined to determine whether or not it is a supervisor call instruction associated with an input/output control macro instruction. If it is found, as the result of the examination, to be a supervisor call instruction for an input/output control instruction, the operating system 23 for the emulated program 22 is not interrupted and the emulation mode is released, then an interrupt is caused in the interface program, that is, ECP 21 which operates in the native mode. The ECP 21 analyses and translates the input/output control instruction issued from the emulated program 22 into an input/output control macro instruction for the native mode operating system 10, then issues the obtained instruction.

DEPR:
The flowchart of FIG. 3 covers the operations of the emulation microprogram up to the translation of an input/output control macro instruction.

DEPR:
If the emulation program detects a supervisor call (SVC) instruction while sequentially executing machine language codes of the target machine, it references the area 414 in the local execution list (emulation mode control table) 41 established when the emulator is initiated (FIG. 4), and obtains one of the table entry, the first address of the SVC jump table 42 (steps 31 and 33). Then the microprogram references a value stored in a byte at an address corresponding to a combination of the obtained first address and the SVC code (0-255 indicating the operand value of SVC instruction) of the jump table 42. This value of SVC jump table 42 has been coded in advance by checking the SVC code in the target machine operating system 23 and contains (FF)16 for an entry corresponding to an SVC code which is associated with an input/output control instruction or a value other than (FF)16 for other entry. The emulation microprogram tests the entry value of the table 42. If the value is other than (FF)16 corresponding to an SVC code, it regards the instruction as an SVC instruction that can be processed on the target machine operating system 23 and carries out the emulation to perform SVC operation as predetermined in accordance with the target machine architecture (steps 34-35). If the entry value is (FF)16, the emulation microprogram regards the instruction as an SVC instruction which is associated with an input/output control macro instruction, saves the PSW indicating the current emulation mode state to the PSW save area 411 in the local execution list 41 depicted in FIG. 4, and at the same time, loads the new ECP interrupt PSW (412) indicating the entry address of the EPC 21 for translating the input/output control macro instruction of the target machine into the PSW, then releases the emulation mode and transfers control to the ECP 21 (steps 36-39).

DEPR:
The ECP 21 can obtain the SVC instruction address by referencing the local PSW 412, hence it is able to read a group of parameters related to the input/output control which is associated with the SVC instruction and thus can easily perform the translation to obtain an input/output macro instruction for the native machine operating system 10.

CLPV:
a. detecting an interrupt instruction defined in a first program in which an instruction of said target machine is executed;

CLPV:
b. determining whether or not said interrupt instruction is an input/output control macro instruction by use of an operand value of said interrupt instruction; and

CLPV:
c. if said interrupt instruction is determined to be an instruction associated with an input/output control macro instruction in said step b, bypassing the operation system of said target machine by translating said input/output control macro instruction from said first program into an input/output control macro instruction for a second program, in which an instruction of said native machine is executed, under control of a program for controlling said emulation.

CLPV:

d. causing an interrupt in an operating system for said first program is said interrupt instruction is not determined to be an instruction associated with an input/output control macro instruction in said step b.

CLPV:
a. detecting an interrupt instruction existing in a first program in which an instruction of said target machine is executed;

CLPV:
b. searching a table by use of an operand of said interrupt instruction;

CLPV:
e. loading said program status word as an interrupt in an emulation program; and

CLPV:
f. releasing said emulation mode and bypassing the operating system of said target machine by translating an input/output control macro instruction from said first program into an input/output control macro instruction for a second program in which an instruction of said native machine is executed.

Full | Title | Citation | Front | Review | Classification | Date | Reference | Claims | KWIC | Draw Desc | Image

Generate Collection

Term	Documents
PIPELINE.USPT.	28264
PIPELINES.USPT.	12959
INTERRUPT$	0
INTERRUPT.USPT.	80738
INTERRUPTA.USPT.	1
INTERRUPTABILITY.USPT.	38
INTERRUPTABLE.USPT.	1063
INTERRUPTABLE-POWER.USPT.	2
INTERRUPTABLE-TYPE.USPT.	1
"INTERRUPTABLE.SUB".USPT.	1
(L1 AND (PIPELINE OR INTERRUPT$ OR TRANSLAT$ OR (TABLE ADJ LOOKUP))).USPT.	4

There are more results than shown above. Click here to view the entire set.

Display 10 Documents, starting with Document: 4

Display Format: KWIC Change Format

WEST

Generate Collection

Search Results - Record(s) 1 through 1 of 1 returned.

□ 1. Document ID: US 5751982 A

L3: Entry 1 of 1 File: USPT May 12, 1998

DOCUMENT-IDENTIFIER: US 5751982 A
TITLE: Software emulation system with dynamic translation of emulated instructions for increased processing speed

ABPL:
The execution time overhead of software emulation is reduced by selecting frequently emulated instruction sequences in the software being emulated, and translating those instruction sequences into the instruction set of the emulating processor. In a first phase, frequently repeated sequences of emulated computer code are identified and selected for translation. In a second phase, the selected sequences from the instruction set of the emulated processor are translated into equivalent sequences for the instruction set of the emulating processor. In a third phase, the instruction sequence of the emulating processor is executed in lieu of emulating the original instructions from the emulated software.

BSPR:
In accordance with the present invention, the processing overhead of an emulating system is reduced by selecting frequently emulated instruction sequences in the software being emulated and translating those instruction sequences into the native instruction set of the emulating processor. The dynamic translation process of the present invention is implemented in three main phases. In the first phase, an identification is made of those code sequences in the emulated software that are frequently repeated. This can be done, for example, by recording all program counter values that are produced by instructions that cause a non-sequential change in the value of the program counter for the emulated software. On a periodic basis, the recorded program counter values are analyzed, to identify code sequences which are emulated frequently enough to warrant dynamic translation. The program counter values which are selected identify the starting point of emulated code sequences to be translated.

BSPR:
In the second phase, the selected code sequences are translated from the instruction set of the emulated processor into the instruction set of the emulating processor. For each emulated instruction in a selected code sequence, its equivalent code sequence in the native instruction set is obtained from the emulator's set of semantic routines, by indexing into the dispatch table with a binary code for the emulated instruction. The successively retrieved code sequences are cumulatively stored in an instruction buffer, until each instruction in the selected sequence has been translated.

BSPR:
In the final phase, the translated sequence is executed in place of emulating the untranslated sequence. This is done by assigning each newly translated code sequence an available operation code from the emulated instruction set. The start of the untranslated instruction sequence is replaced with the value of this operation code, and the address of the translated instruction sequence is inserted in the dispatch table as the corresponding entry for this operation code. From that point on, the emulation software automatically executes the translated code sequence in lieu of emulating the untranslated code sequence, resulting in a substantial improvement in the time required to perform the emulation.

BSPR:
As a further feature of the invention, the computation of gratuitous values that are not employed during execution, such as flag values for emulated condition code registers, can be eliminated to further optimize the translated code sequence. In the implementation of this feature, an analysis of the emulated code sequence being translated is performed to identify any emulated instructions in the sequence which unnecessarily modify emulated condition flags. The modification of an emulated condition flag is unnecessary when a subsequent emulated instruction in the sequence modifies the same flag and no intervening emulated instruction either refers to the flag value or causes a non-sequential change to the emulated program counter value. For each emulated instruction where these conditions are met, an alternative native code sequence, which does not include the instructions to compute the unnecessary condition flag values, is employed in the translation of that emulated instruction.

DRPR:
FIG. 2 is a block diagram similar to FIG. 1, which depicts the conventional translation of a code block;

DRPR:
FIG. 4 is a block diagram depicting the execution of a translated code block.

DEPR:
Referring to FIG. 2, an example of a code block is represented by the shaded area 22. In a conventional emulation process, the instructions in a code block are individually translated each time they are encountered during the emulation process. As each instruction is fetched by the dispatcher 14, it functions as an address to the dispatch table 16. The dispatcher 14 retrieves the pointer stored at the addressed portion of the table 16, and calls the corresponding semantic routine to which the pointer refers, for execution. In the specific example of FIG. 2, the code block 22 consists of three instructions. As is conventional in most programs, each instruction is represented by a unique numerical value, known as its operation code, or opcode. For this example, the three instructions have the opcodes 2030, 2031 and 4e75. These three instructions respectively result in three dispatch operations to separately retrieve and execute three semantic routines for those opcodes, respectively. This need to continually dispatch each instruction as it is issued by the application program presents a significant amount of processing overhead in the overall operation of the emulator.

DEPR:
In accordance with the present invention, this processing overhead can be considerably reduced through dynamic translation of selected code blocks in the emulated application program. To implement this feature, the code blocks which are emulated frequently enough to warrant dynamic translation are first identified. This can be carried out by recording program counter values that produce non-sequential changes in the sequence of instructions being emulated. Whenever an instruction from the CISC code is emulated that results in a non-sequential change to the program counter, the new program counter value is recorded, for example by pushing its value onto a programmatic stack. In essence, each recorded program counter value represents the starting point of a new code block. Whenever there is a break in the operation of the emulator, for example as the processor stops emulation to service a special event or an interruption, the accumulated values are removed from the stack and analyzed to identify code blocks that are emulated more than a defined number of times within a predetermined time window. For example, if a particular block is emulated more than 256 times within a period of about 16 milliseconds, it may be selected. Any suitable approach can be employed to select the program counter values that identify the code blocks which occur with sufficient frequency. In the preferred embodiment of the invention, as each recorded program counter value is removed from the stack, its value is used as a hash index into a table of frequency counts, and the corresponding entry in the table is incremented by one. When the count is incremented beyond a predetermined threshold value, the program counter value corresponding to that table entry is placed on a list of code blocks to be dynamically translated.

DEPR:
The selected values identify the starting points of emulated code sequences, or code blocks, that are to be dynamically translated. Each code block begins with an instruction that is the target of a non-sequential change in the value of the

program counter, and ends with the last instruction that can be reached by any code path that is wholly contained within the code block. Once a code block has been selected for dynamic translation, each instruction in the selected code block is translated from the instruction set of the emulated processor into the native code. For each instruction in the code block, its equivalent semantic routine is identified in the native code, using the dispatch table 16. Referring to FIG. 3, as the first instruction in the code block, in this example 2030, is translated, its corresponding semantic routine 2030 is stored in a buffer 26. As the next instruction in the code block is translated, its semantic routine 2031 is appended to the instructions stored in the buffer 26. This process is repeated until each instruction in the selected code block has been translated and its corresponding semantic routine stored in the instruction buffer 26.

DEPR:
This entire dynamic translation procedure, namely (a) the analysis of program counter values, (b) the identification of frequently emulated code blocks, and (c) the translation and storage of selected code blocks in the buffer, preferably takes place during the interruption of the emulation, i.e. prior to the time that the event which interrupted the emulation is serviced.

DEPR:
Once the entire code block has been translated in this manner, the translated sequence is executed in place of emulating the individual instructions of the untranslated sequence. To do so, the newly translated code sequence is assigned an opcode from the instruction set of the emulated processor. For example, the instruction set for the Motorola 68000 series of microprocessors employs a 16-bit opcode. This presents the capability for over 65,536 different possible opcodes that can be represented. Of this number, about 50,000 are actually employed for the instruction set, leaving a remainder of about 15,000 unassigned operation codes that are available for other uses. In the implementation of the present invention, each newly translated code sequence that is stored in an instruction buffer 26 is assigned one of the available operation codes from the emulated instruction set. The beginning instruction of the untranslated instruction sequence 22 is overwritten with the value of this new, unassigned operation code. Referring to the example of FIG. 4, the opcode 6001 is assumed to be one such unassigned, and hence available, opcode. It is substituted for the opcode 2030 of the first instruction in the code block 22. In the dispatch table 16, the address of the translated instruction sequence in the buffer 26 is inserted as the corresponding entry for the new operation code. Subsequently, whenever the application program issues the first instruction in the translated code block, the dispatcher 14 will point to the translated code sequence, which will be automatically executed by the emulation software, in lieu of emulating the untranslated code sequence. Consequently, only a single call to the dispatcher is made, rather than a separate call for each instruction in the code block, resulting in a substantial improvement in the time required to perform the emulation.

DEPR:
As a feature of the present invention, the performance of the emulator can be further improved by eliminating the computation of gratuitous condition code values under those conditions in which their elimination results in semantically identical results in the emulation of the code block. This feature of the invention can be implemented during the second phase of the dynamic translation process, i.e. while the instructions of the selected code block are being translated. During this phase, the instructions in the emulated code are analyzed to identify those situations in which a subsequent instruction, or set of instructions, modifies all of the condition code flags that are set by the instruction under analysis. If no intervening instructions refer to the condition code flags, or cause a potential non-sequential change in the program counter value, the computation of the condition code flags can be eliminated. In this situation, an alternative semantic routine, which does not include computation of the condition code flag values, is employed for the translation of the instruction under analysis. The alternate semantic routine for an emulated instruction can be stored immediately following the primary semantic routine for that instruction. In the case of some CISC instructions, the elimination of condition code flag computations can reduce the number of native instructions in the semantic routine by two-thirds, resulting in appreciable speed and code size savings.

DEPR:
From the foregoing, it can be seen that the present invention employs dynamic translation to significantly improve the execution time of a software emulator. The dynamic translation results in a single dispatch operation for an entire code block of the emulated code, thereby significantly reducing the dispatch overhead associated with the emulation process. In the implementation of the invention, it is desirable to translate as many instructions as possible in a code block. The longer the sequence of instructions that are dynamically translated, the greater the increase in the emulation speed. This increased performance is due to the elimination of instruction decoding overhead, as well as increased locality within the target processor's instruction caches. More particularly, the sequence of native instructions that comprises a translated code block will generally fit within the available instruction cache memory of a RISC processor. When the translated block contains frequently iterated loops of code, a large stream of instructions will be executed directly out of cache memory at a much greater overall instruction throughput. Conversely, if that same block were to be emulated, there will likely be cache conflicts generated between the emulator's dispatcher, the dispatch table, and the semantic routines, and these delays will be multiplied by the number of loop iterations within the block.

Full | Title | Citation | Front | Review | Classification | Date | Reference | Claims | KWIC | Draw. Desc | Image

Generate Collection

Term	Documents
CODE.USPT.	176960
CODES.USPT.	81798
TRANSLAT$	0
TRANSLAT.USPT.	11
TRANSLATABALE.USPT.	1
TRANSLATABILITY.USPT.	98
TRANSLATABLE.USPT.	6991
TRANSLATABLE-BULB.USPT.	1
TRANSLATABLE/ADJUSTABLE.USPT.	3
TRANSLATABLE/LOCKING.USPT.	1
(L1 AND (CODE WITH TRANSLAT$)).USPT.	1

There are more results than shown above. Click here to view the entire set.

Display □ Documents, starting with Document: □

Display Format: □ Change Format

WEST

Your wildcard search against 2000 terms has yielded the results below

Search for additional matches among the next 2000 terms

Generate Collection

Search Results - Record(s) 1 through 3 of 3 returned.

□ 1. Document ID: US 6006029 A

L4: Entry 1 of 3 File: USPT Dec 21, 1999

DOCUMENT-IDENTIFIER: US 6006029 A
TITLE: Emulating disk drives of a first system on a second system

DRPR:
FIG. 8 is the address translation mechanism and memory space mapping mechanism of the emulation mechanism.

DEPR:
5. Addresses and Address Translation

DEPR:
It will be noted, as described previously, that Software Active Queue (SAQ) 88, the Pseudo Device Queues (PSDQs) 86, and INTERPRETER 72 are provided to emulate the corresponding mechanisms of First System 10, that is, First System 10's input/output devices and central processing unit, as seen by Executive Program Tasks (EXP Tasks) 28 and Tasks 30. As such, Executive Program Tasks (EXP Tasks) 28 and Tasks 30 will provide memory addresses to the Pseudo Device Queues (PSDQs) 82 and INTERPRETER 72 according to the requirements of the native memory access and management mechanisms of First System 10 and will expect to receive memory addresses from Software Active Queue (SAQ) 88 and INTERPRETER 72 in the same form. Second System Kernel Processes (SKPs) 66, Lower Communications Facilities Layer Processes (LCFLPs) 78, the hardware elements of Second System 54 and other processes executing as native processes in Second System 54, however, operate according to the memory addressing mechanisms native to Second System 54. As such, address translation is required when passing requests and returning requests between Emulator Executive Level (EEXL) 68 and Second System Kernel Level (SKernel) 64.

DEPR:
As described, INTEPRETER 70 is provided to interpret First System 10 instructions into functionally equivalent Second Second 54 instructions, or sequences of instructions, including instructions pertaining to memory operations. As such, the address translation mechanism is also associated with INTERPRETER 72, or is implemented as a part of INTERPRETER 72, and is indicated in FIG. 3 as Address Translation (ADDRXLT) 98 and will be described in detail in a following discussion.

DEPR:
As described above with reference to FIGS. 2 and 3, the First System 10 tasks and programs executing on Second System 54, Second System 54's native processes and mechanisms and the Second System 54 mechanisms emulating First System 10 mechanisms share and cooperatively use Second System 54's memory space in Second System Memory 58b. As a consequence, it is necessary for Second System 54, the First System 10 tasks and programs executing on Second System 54, and the emulation mechanisms to share memory use, management, and protection functions in a manner that is compatible with both Second System 54's normal memory operations

and with First System 10's emulated memory operations. The emulation of First System 10 memory operations in Second System 54 in turn requires emulation of First System 10's memory management unit, that is, First System 10's hardware and software elements involved in memory space allocation, virtual to physical address translation, and memory protection in Second System 54. As described below, this emulation is implemented through use of Second System 52's native memory management unit to avoid the performance penalties incurred through a complete software emulation of First System 10's memory management unit.

DEPR:
As is well known, most systems operate upon the basis of virtual addresses and perform virtual to physical address translations relative to a predetermined base address, that is, by adding a virtual address as an offset address to the base address to determine the corresponding address in physical address space of the system. While First System 10 and Second System 52 may both use such addressing schemes, the actual addressing mechanisms of the two system may differ substantially, as may the memory protection schemes.

DEPR:
Referring to FIG. 8, and to FIGS. 2, 3, 5 and 6, therein is illustrated the mechanisms implemented on Second System 54 to emulate the memory access, protection, and management mechanisms of First System 10. It must be recognized in the following that the emulation of First System 10 memory operations on Second System 54 involves two differerent address conversion operations, one being the conversion of First System Virtual Addresses (FSVAs) 126 done by INTERPRETER 72 and the second being the conversion of First System Virtual Addresses (FSVAs) 126 done by Pseudo Device Drivers (PSDDs) 74. Each of these conversions is accomplished through translation and through mapping of First System 10 system and user memory areas into Second System 54 segments. The following will first describe the address translation operation performed by INTERPRETER 72, and then will describe the address translation operation performed by Pseudo Device Drivers (PSDDs) 74.

DEPR:
First considering the process of INTERPRETER 72 address translation, as has been described above, each First System Virtual Address (FSVA) 126 is comprised of a Most Significant Bits field (MSB) 128 and an Address field (ADDR) 130 wherein Most Sigificant Bits field (MSB) 128 contains a bit field whose value identifies whether the address is directed to an executive memory area, that is, System Memory (SYSMEM) 110 area, or to an Independent-Memory Pool (IPOOL) 112. For example, the Most Sigificant Bits field (MSB) 128 may contain the value 0000 (0) when the request is directed to the System Memory (SYSMEM) 110 area and the value 0001 (1) when the request is directed to an Independent-Memory Pool (IPOOL) 112 area.

DEPR:
As indicated in FIG. 8, the First System Virtual Address (FSVA) 126 of a request which includes a memory access is provided to Address Translation (ADDRXLT) 98. Address Translation (ADDRXLT) 98 includes a Word To Byte Shifter (WBS) 148 which performs an initial translation of the First System Virtual Address (FSVA) 126 from the First System 10 format, in which addresses are on a per word basis, to a Second System 54 virtual address, in which addresses are on a per byte basis. This translation is performed by a left shift of the First System Virtual Address (FSVA) 126 and, in the translation and as indicated in FIG. 7, the value in the Most Sigificant Bits field (MSB) 128 field of the First System Virtual Address (FSVA) 126 is transformed from 0000 (0) or 0001 (1) to 0000 (0) or 0010 (2), respectively.

DEPR:
Having performed the translation of a First System Virtual Address (FSVA) 126 into a per byte address, Address Translation (ADDRXLT) 98's Ring Adder (RNGA) 150 will read a System Status Register (SSR) 152 which, among other information, contains a Ring Number (RNG) 154 which contains a value indicating the First System 10 ring in which the task is executing, that is, a value of 0, 1, 2 or 3. As described, Ring 0 is reverved for system operations while Rings 1, 2 and 3 are used for user tasks. If the task is executing in Ring 0, that is, in system space, Ring Adder (RNGA) 150 will add 3 to the value (0 or 2) contained in Most Significant Bits field (MSB) 128 of the shifted First System Virtual Address (FSVA) 126. If the task is not executing in Ring 0, that is, is executing in

Rings 1, 2, or 3 and thus in user task space, Ring Adder (RNGA) 150 will add 4 to the value (0 or 2) contained in Most Significant Bits field (MSB) 128 of the shifted First System Virtual Address (FSVA) 126. The final result will be a byte oriented First System Virtual Address (FSVA) 126 having a Most Significant Bits field (MSB) 128 which contains a value of 3, 4, 5 or 6, thereby indicating the Second System 54 memory space segment in which the address lies and an Address (ADDR) field 130 identifying a location within the segment.

DEPR:
The INTERPRETER 72 translation process always generates adddresses in segments 5 and 6 for user task addresses, but because of dynamic detaching and attaching of Independent Memory Pools (IPOOLs) 112, the same addresses will refer to different Independent Memory Pools (IPOOLs) 112. The mapping of system memory areas remains the same, however, when switching from Task Control Block (TCB) 32 to Task Control Block (TCB) 32, so that the INTERPRETER 72 generated addresses in segments 3 and 4 always refer to the same locations.

DEPR:
As has been described, each Pseudo Device Driver Queue (PSDQ) 86 is associated with a corresponding Second System Kernel Process (SKP) 66 which executes the requests in the Pseudo Device Driver Queue (PSDQ) 86 and any Pseudo Device Driver Queue (PSDQ) 86 may contain requests from a plurality of tasks, each task in turn being associated with and executed in an Independent-Memory Pool (IPOOL) 112 area which is mapped into a Second System 54 memory segment by address translator (ADDRXLP) 96 which includes a Server Pool Descriptor Linked Set (SPDLS) associated with the Pseudo Device Driver Queue (PSDQ) 86, Task Control Block (TCB) 32, Segment Descriptor Table 156, and Memory Pool Array 162.

DEPR:
It may be seen from the above descriptions, therefore, that, for any first system virtual address generated by a First System 10 task executing on Second System 54, INTERPRETER 72 will translate the First System 10 virtual address into a byte oriented virtual address containing a virtual address location within a segment and identifying a Segment 3, 4, 5 or 6 containing the location. The INTERPRETER 72 mapping of segments via ADDRXLT98 will in turn map each segment identified by an address translation into an Independent Memory Pool Identification (IPOOLID) 160 for the current task. The Segment/Independent Memory Pool mapping mechanism (i.e., ADDRXLP96) of the Pseudo Device Driver (PSDD) 74 executing the task request associated with the First System 10 virtual address will map the segment identified by the address translation mechanism to a current Independent Memory Pool (IPOOL) 112 location in System 54's memory by providing the base address corresponding to the Independent Memory Pool Identification (IPOOLID) 160.

Full | Title | Citation | Front | Review | Classification | Date | Reference | Claims | KWIC | Draw Desc | Image

□ 2. Document ID: US 5751982 A

L4: Entry 2 of 3 File: USPT May 12, 1998

DOCUMENT-IDENTIFIER: US 5751982 A
TITLE: Software emulation system with dynamic translation of emulated instructions for increased processing speed

BSPR:
In the final phase, the translated sequence is executed in place of emulating the untranslated sequence. This is done by assigning each newly translated code sequence an available operation code from the emulated instruction set. The start of the untranslated instruction sequence is replaced with the value of this operation code, and the address of the translated instruction sequence is inserted in the dispatch table as the corresponding entry for this operation code. From that point on, the emulation software automatically executes the translated code sequence in lieu of emulating the untranslated code sequence, resulting in a substantial improvement in the time required to perform the emulation.

DEPR:
Once the entire code block has been translated in this manner, the translated sequence is executed in place of emulating the individual instructions of the untranslated sequence. To do so, the newly translated code sequence is assigned an opcode from the instruction set of the emulated processor. For example, the instruction set for the Motorola 68000 series of microprocessors employs a 16-bit opcode. This presents the capability for over 65,536 different possible opcodes that can be represented. Of this number, about 50,000 are actually employed for the instruction set, leaving a remainder of about 15,000 unassigned operation codes that are available for other uses. In the implementation of the present invention, each newly translated code sequence that is stored in an instruction buffer 26 is assigned one of the available operation codes from the emulated instruction set. The beginning instruction of the untranslated instruction sequence 22 is overwritten with the value of this new, unassigned operation code. Referring to the example of FIG. 4, the opcode 6001 is assumed to be one such unassigned, and hence available, opcode. It is substituted for the opcode 2030 of the first instruction in the code block 22. In the dispatch table 16, the address of the translated instruction sequence in the buffer 26 is inserted as the corresponding entry for the new operation code. Subsequently, whenever the application program issues the first instruction in the translated code block, the dispatcher 14 will point to the translated code sequence, which will be automatically executed by the emulation software, in lieu of emulating the untranslated code sequence. Consequently, only a single call to the dispatcher is made, rather than a separate call for each instruction in the code block, resulting in a substantial improvement in the time required to perform the emulation.

Full | Title | Citation | Front | Review | Classification | Date | Reference | Claims | KWIC | Draw. Desc | Image

☐ 3. Document ID: US 4812975 A

L4: Entry 3 of 3 File: USPT Mar 14, 1989

DOCUMENT-IDENTIFIER: US 4812975 A
TITLE: Emulation method

DEPR:
If the emulation program detects a supervisor call (SVC) instruction while sequentially executing machine language codes of the target machine, it references the area 414 in the local execution list (emulation mode control table) 41 established when the emulator is initiated (FIG. 4), and obtains one of the table entry, the first address of the SVC jump table 42 (steps 31 and 33). Then the microprogram references a value stored in a byte at an address corresponding to a combination of the obtained first address and the SVC code (0-255 indicating the operand value of SVC instruction) of the jump table 42. This value of SVC jump table 42 has been coded in advance by checking the SVC code in the target machine operating system 23 and contains (FF)16 for an entry corresponding to an SVC code which is associated with an input/output control instruction or a value other than (FF)16 for other entry. The emulation microprogram tests the entry value of the table 42. If the value is other than (FF)16 corresponding to an SVC code, it regards the instruction as an SVC instruction that can be processed on the target machine operating system 23 and carries out the emulation to perform SVC operation as predetermined in accordance with the target machine architecture (steps 34-35). If the entry value is (FF)16, the emulation microprogram regards the instruction as an SVC instruction which is associated with an input/output control macro instruction, saves the PSW indicating the current emulation mode state to the PSW save area 411 in the local execution list 41 depicted in FIG. 4, and at the same time, loads the new ECP interrupt PSW (412) indicating the entry address of the EPC 21 for translating the input/output control macro instruction of the target machine into the PSW, then releases the emulation mode and transfers control to the ECP 21 (steps 36-39).

DEPR:
The ECP 21 can obtain the SVC instruction address by referencing the local PSW 412, hence it is able to read a group of parameters related to the input/output control which is associated with the SVC instruction and thus can easily perform the translation to obtain an input/output macro instruction for the native machine operating system 10.

Full | Title | Citation | Front | Review | Classification | Date | Reference | Claims | KWIC | Draw Desc | Image

Generate Collection

WEST

Generate Collection

Search Results - Record(s) 1 through 4 of 4 returned.

□ 1. Document ID: US 6006029 A

L7: Entry 1 of 4 File: USPT Dec 21, 1999

DOCUMENT-IDENTIFIER: US 6006029 A
TITLE: Emulating disk drives of a first system on a second system

DEPR:
The operations performed in First System 10 in execution of an Application Program (APP) 22 or a System Administrative Program (SAD) 24 are executed through a plurality of Tasks 30 and any program executing on First System 10 may spawn one or more Tasks 30. A Task 30 may be regarded as being analogous to a process, wherein a process is generally defined as a locus of control which moves through the programs and routines and data structures of a system to perform some specific operation or series of operations on behalf of a program. There is a Task Control Block (TCB) 32 associated with each Task 30 wherein the Task Control Block (TCB) 32 of a Task 30 is essentially a data structure containing information regarding and defining the state of execution of the associated Task 30. A Task Control Block (TCB) 32 may, for example, contain information regarding the state of execution of tasks or operations that the Task 30 has requested be performed and the information contained in a Task Control Block (TCB) 32 is available, for example, to the programs of Executive Program Tasks (EXP Tasks) 28 for use in managing the execution of the Task 30. Each Task 30 may also include an Interrupt Save Area (ISA) 34 which is used to store hardware parameters relevant to the Task 30.

DEPR:
The Queue Frames (QFs) 94 of Software Active Queue (SAQ) 88 and Pseudo Device Driver Queues (PSDQs) 86 differ in detail and the following will describe the Queue Frames (QFs) 94 of both, noting where the frames differ. Each Queue Frame (QF) 94 further includes a Task Control Block Pointer (TCBP) or Input/Output Request Block Pointer (IORBP) 38p, as previously described, a Priority Field (Priority) 108 containing a value indicating the relative priority of the interrupt or request. The Queue Frames (QFs) 94 of Software Active Queue (SAQ) 88 include a Flag Field (Flag) 108 containing a flag which distinguishes whether the Queue Frame (QF) 94 contains a Task Control Block (TCB) 32 or an Indirect Request Block (IRB) 36. Input/Output Request Blocks (IORBs) through their IRBs are generally given a higher priority than Task Control Blocks (TCBs). Exceptions may be made, however, for example, for clock and task inhibit Task Control Blocks (TCBs) 32, which must be given the highest priority.

Full | Title | Citation | Front | Review | Classification | Date | Reference | Claims | KWIC | Draw Desc | Image

□ 2. Document ID: US 5751982 A

L7: Entry 2 of 4 File: USPT May 12, 1998

DOCUMENT-IDENTIFIER: US 5751982 A
TITLE: Software emulation system with dynamic translation of emulated instructions for increased processing speed

DEPR:
In accordance with the present invention, this processing overhead can be considerably reduced through dynamic translation of selected code blocks in the emulated application program. To implement this feature, the code blocks which are emulated frequently enough to warrant dynamic translation are first identified. This can be carried out by recording program counter values that produce non-sequential changes in the sequence of instructions being emulated. Whenever an instruction from the CISC code is emulated that results in a non-sequential change to the program counter, the new program counter value is recorded, for example by pushing its value onto a programmatic stack. In essence, each recorded program counter value represents the starting point of a new code block. Whenever there is a break in the operation of the emulator, for example as the processor stops emulation to service a special event or an interruption, the accumulated values are removed from the stack and analyzed to identify code blocks that are emulated more than a defined number of times within a predetermined time window. For example, if a particular block is emulated more than 256 times within a period of about 16 milliseconds, it may be selected. Any suitable approach can be employed to select the program counter values that identify the code blocks which occur with sufficient frequency. In the preferred embodiment of the invention, as each recorded program counter value is removed from the stack, its value is used as a hash index into a table of frequency counts, and the corresponding entry in the table is incremented by one. When the count is incremented beyond a predetermined threshold value, the program counter value corresponding to that table entry is placed on a list of code blocks to be dynamically translated.

DEPR:
This entire dynamic translation procedure, namely (a) the analysis of program counter values, (b) the identification of frequently emulated code blocks, and (c) the translation and storage of selected code blocks in the buffer, preferably takes place during the interruption of the emulation, i.e. prior to the time that the event which interrupted the emulation is serviced.

CLPR:
5. The method of claim 4 wherein said counting step comprises the steps of loading a value associated with said non-sequential change in a stack upon each detected occurrence, emptying said stack in response to an interruption in an emulation operation, and counting the number of occurrences of each value in said stack.

| Full | Title | Citation | Front | Review | Classification | Date | Reference | Claims | KWIC | Draw Desc | Image |

□ 3. Document ID: US 5678032 A

L7: Entry 3 of 4 File: USPT Oct 14, 1997

DOCUMENT-IDENTIFIER: US 5678032 A
TITLE: Method of optimizing the execution of program instuctions by an emulator using a plurality of execution units

DEPR:
In the emulated system, each task 30 utilizes a plurality of data control structures, such as a task control block (TCB) structure 32, an indirect request block (IRB) structure 36, an input/output request block (IORB) structure 38 and a resource control table (RCT) structure 40. The task control block (TCB) structure 32 contains information pertaining to the state of execution of the associated task as web as pointers to interrupt save areas for storing hardware parameters

related to the task. The indirect request block (IRB) structure 36 contains information defining the operation requested by an associated task and includes pointers identifying the task and its associated task control block (TCB) and a pointer to the associated IORB structure.

DEPR:
As indicated in FIG. 1b, the next layer within the user level is the emulator executive level 68. This level includes certain components present in the emulated system which have been transformed into new mechanisms which appear to the remaining unchanged components to operate as the original unchanged components of the emulated system. At the same time, these new mechanisms appear to the components of the kernel level 64 as native components with which the host system is accustomed to operate, As shown, the components include the interpreter 72, an emulator monitor call unit (EMCU) 73, an emulator interrupt handler unit (EEHU) 74, a plurality of servers 90 through 98, and a plurality of pseudo device drivers (PSDD) 86 arranged as shown.

DEPR:
The kernel process manager component 70 includes an exception/interrupt services mechanism. The mechanism handles both interrupts and exception conditions (i.e. unexpected events caused by a process such as illegally addressing memory which can cause a segment violation signal). Exceptions can happen "in the middle" of the execution of a RISC instruction and the host system normally attempts to restart the instruction after handling the exception condition.

DETL:

APPENDIX I BLOCK F # Start of Instruction # (normal start, bypassed in optimized code) soi: sli. dt,iw,2 # iw * 4 lx w0,dt,top, # fetch XA entry soi.sub.-- p2: mtlr w0 # prepare to splatter bge 0,soi.sub.-- 0 # branch for opcodes 0-0x7ff (executed by ICU) rlinm w2,iw,28,21,28 # get index to XE table soi.sub.-- 1: P1 lux wo,w2,xep # fetch XE pointer (MAS) (f) Q1 lha dt,2(p) # fetch @ (P + 2) (next instr. if current instr. is one word) (b) P2 l w1,4(w2) # fetch XE pointer (RAS) (f) Q2 lha z,4(p) # fetch @ (P + 4) (next inst. if current inst. is two words) (b) (start of instruction, optimized code) soi.sub.-- 1b: P3 mtctr w0 # move memory AS XE address to CTR (f) Q3 sli. dt,dt,2 # [@(P + 2)]*4 (b) P4 st p,ph # save copy of P for traps (f) Q4 sli z,z,2 # [@(P + 4)] * 4 (b) bbfr 12 # jump to entry pt if no interrupt (f) (go to XA fragment/BLOCK D) BLOCK D # Base register effective address calculation, operand fetch P5 mr ea,bl # centralize b register (f) Q5 lx w0,top,dt # fetch next IW XA (b) P6 mr z,dt # centralize next IW (b) bbtl 15,.sub.-- xvba.sub.-- vp # if v pool, go to addr convert routine (f) Q6 lha dt,0(ea) # fetch word operand (f) P7 cal p,2(p) # advance P to next IW (b) bctr # splatter to XE (f) (continue in XE fragment/BLOCK E) BLOCK E # Load R register execution routine P8 srai iw,z,2 # set next IW (b) Q8 mtlr w0 # prepare to splatter (b) P9 mr rl,dt # move operand to register (f) Q9 rlinm w2,iw,28,21,28 # get index to XE table (f) bge 0,soi.sub.-- 0 # branch for opcodes 0-0x7fff (f) P10 lha dt,2(p) # fetch IW + 2 (b) Q10 lux w0,w2,xep # fetch XE pointer (MAS) (f) P11 lha z,4(p) # fetch IW + 4 (b) Q11 l w1,4(w2) # fetch XE pointer (RAS) (f) b soi.sub.-- 1b # return to fetch routine (f) (return to start of instruction/BLOCK F, label APPENDIX II BLOCK F # # Start of Instruction # (normal start, bypassed in optimized code) soi: sli. dt,iw,2 # iw * 4 lx w0,dt,top # fetch XA entry soi.sub.-- p2: mtlr w0 # prepare to splatter bge 0,soi.sub.-- 0 # branch for opcodes 0-0x7fff rlinm w2,iw,28,21,28 # get index to XE table (start of instruction, optimized code) soi.sub.-- 1: P1 lux w0,w2,xep # fetch XE ptr (MAS) (f) Q1 lha dt,2(p) # fetch @(P + 2) (b) P2 l w1,4(w2) # fetch XE ptr (RAS) (f) Q2 lha z,4(p) # fetch @(P + 4) (b) soi.sub.-- 1b: P3 mtctr w0 # move memory AS XE addr to CTR (f) Q3 sli. dt,dt,2 # [@(P + 2)] * 4 (b) P4 st p,ph # save copy of P for traps (f) Q4 sli z,z,2 # [@(P + 4)] * 4 (b) bbfr 12 # jump to entry point if no intr (f) (go to XA fragment/BLOCK D) BLOCK D # Base register + index register # effective address calculation, operand fetch P5 cax ea,bl,r2 # add index to base (f) Q5 mr z,dt # centralize next IW (b) P6 lx w0,top,z # fetch next IW XA (b) bbtl 15,.sub.-- xvba.sub.-- vp # if v pool, go to addr convert routine (f) Q6 lha dt,0(ea) # fetch word operand (f) Segment Violation occurs here, continue in Signal Catcher C code Segment Violation Signal Catcher C Code (partial) /* handle rare case of word prefetch at last byte of AIX segment */ /* case 1: I pool, multiple sources of segvio */ /* test lower 12 bits of register ea for all 1 (last byte of page) */ if((scp->sc.sub.-- jmpbuf.jmp.sub.-- context.gpr[25] & 0x0fff) = 0xfff) { /* test instruction is : `lha dt,0(ea)`*/ longbase = (long *) scp->sc.sub.-- jmpbuf.jmp.sub.-- context.iar; if (*longbase = 0xab190000) { /*

set link register to follovang instruction */ scp->sc.sub.-- jmpbuf.jmp.sub.-- context.lr = scp->sc.sub.-- jmpbuf.jmp.sub.-- context.iar+4; /* direct interpreter to fixup code */ scp->sc.sub.-- jmpbuf.jmp.sub.-- context.iar=(long)fetch8; return; } } /* case 2: V pool, single instance */ if(scp->sc.sub.-- jmpbuf.jmp.sub.-- context.iar = (long) xva16.sub.-- segvio) { scp->sc.sub.-- jmpbuf.jmp.sub.-- context.lr=scp->sc.sub.-- jmpbuf.jmp.sub. -- context.iar+4; scp->sc.sub.-- jmpbuf.jmp.sub.-- context.iar=(long)fetch8v; return; } Assembly code in interpreter unit to refetch byte operand # This code handles extraneous segvio's caused by the # sequence of a byte operand prefetch performed # via a 16 bit read on the very last byte of an AIX # segment. It is continuation code that the interpreter # is directed to by the segvio signal catcher when the # sequence is detected. # .globl fetch8v .globl fetch8 fetch8v: cax ea,ea,dt # add M:seg.sub.-- base to addr fetch8: lbz dt,0(ea) # fetch byte operand sli dt,dt,8 # move to right middle byte exts dt,dt # sign extend br # return to original sequence BLOCK D -(cont'd) Return here after Signal Catcher/fetch 8 code recovery, or continue from BLOCK D (Q7) cal p,2(p) # advance P to next IW (b) bctr # splatter to XE (f) (continue in XE fragment/BLOCK E) BLOCK E # Load R register halfword execution routine (P8) srai iw,z,2 # set next IW (b) (Q8) mtlr w0 # prepare to splatter (b) (P9) srai rl,dt,8 # store left byte (f) (Q9) rlinm w2,iw,28,21,28 # get index to XE table (f) bge 0,soi.sub.-- 0 # branch for opcodes 0-0x7ff (f) (P10) lha dt,2(p) # fetch IW + 2 (b) (Q10) lux w0,w2,xep # fetch XE ptr (MAS) (f) (P11) lha z,4(p) # fetch IW + 4 (b) (Q11) l x1,4(w2) # fetch XE ptr (RAS) (f) b soi.sub.-- lb # return to fetch routine (f) (return to start of instruction/BLOCK F, label)

Full | Title | Citation | Front | Review | Classification | Date | Reference | Claims | KWIC | Draw Desc | Image

□ 4. Document ID: US 4812975 A

L7: Entry 4 of 4 File: USPT Mar 14, 1989

DOCUMENT-IDENTIFIER: US 4812975 A
TITLE: Emulation method

ABPL:

A method for emulating programs in a system includes a plurality of first and second data processors having different instruction word sets. An instruction which interrupts the operating system on the first data processor is defined. When the instruction is detected in a program running on the first data processor, it is determined whether or not the instruction is an instruction associated with an input/output macro instruction. If it is found, as a result of the determination, that this is the case, an interrupt is caused in a program running on the second data processor which controls the emulation, and the input/output macro instruction output from an emulated program is translated into an input/output macro instuction for the operating system, thereby implementing an emulation with a minimized overhead.

BSPR:

The present invention is materialized in a system comprising data processors, for example, a first data processor and a second data processor which have a different instruction word set, respectively. It is assumed that the user programs in the first data processor are processed under control of a first operating system and that those in the second data processor are processed under control of a second operating system. The programs in the first data processor are assigned as the programs to be emulated (to be referred to as emulated programs hereafter) and the specific instructions are defined in these emulated programs. Such an instruction like a supervisor call instruction, is one that causes an interrupt in the first operating system. When.. the interrupt instruction is detected, it is examined to determine whether or not this is an interrupt instruction associated with an input/output control macro instruction. If it is found, as the result of the examination, that this is the case, an interrupt occurs in a program on the second data processor which controls the emulation. Then, the input/output control macro instructions issued by the emulated program are translated into the corresponding input/output control macro instructions for the second operating system.

DEPR:
The EXL instruction format consists of an instruction code part (EXL), a B.sub.2 part comprising four bits for indicating a general-purpose register to be assigned as the base register, and a D.sub.2 part comprising 12 bits which indicate a binary value. When an EXL instruction is issued, the second operand address is register specified by the B.sub.2 part to the D.sub.2 part. The second operand address indicates an address on the main storage device at which a local execution list 41 (FIG. 4) is stored. The local execution list 41 comprises an area 411 for saving the emulation mode PSW necessary for the emulator operation, an area 412 for storing therein the new PSW for the ECP interrupt, a general-purpose register save area 413, an area for storing therein the first address of the jump table for the superviser call (SVC) instruction, and other areas. When the EXL instruction is executed, the pertinent values are set to the PSW, general-purpose register, etc. in the local execution list 41, then the system enters the local execution (emulator operation) mode.

DEPR:
In accordance with the emulation method of the present invention, since an input/output control instruction specified by the user program 24 in the emulated program 22 is issued in the form of an input/output macro instruction of the operating system 23 in the emulated program 22, and it is ordinarily an interrupt instruction, such as an SVC instruction, to the operating system 23; the input/output instruction is emulated at the macro instruction level rather than at the machine language level which has been adopted in the conventional emulation method.

DEPR:
In accordance with the present invention, when an SVC instruction is detected during the execution of the emulated program 22, the micro program for executing the emulation searches the entries of the local execution list 41 (FIG. 4) for the SVC jump table first address entry 414. Each entry of the table 42 is referenced by use of the entry 414 stored in the local execution list 41. Then, the instruction is examined to determine whether or not it is a supervisor call instruction associated with an input/output control macro instruction. If it is found, as the result of the examination, to be a supervisor call instruction for an input/output control instruction, the operating system 23 for the emulated program 22 is not interrupted and the emulation mode is released, then an interrupt is caused in the interface program, that is, ECP 21 which operates in the native mode. The ECP 21 analyses and translates the input/output control instruction issued from the emulated program 22 into an input/output control macro instruction for the native mode operating system 10, then issues the obtained instruction.

DEPR:
If the emulation program detects a supervisor call (SVC) instruction while sequentially executing machine language codes of the target machine, it references the area 414 in the local execution list (emulation mode control table) 41 established when the emulator is initiated (FIG. 4), and obtains one of the table entry, the first address of the SVC jump table 42 (steps 31 and 33). Then the microprogram references a value stored in a byte at an address corresponding to a combination of the obtained first address and the SVC code (0-255 indicating the operand value of SVC instruction) of the jump table 42. This value of SVC jump table 42 has been coded in advance by checking the SVC code in the target machine operating system 23 and contains (FF)16 for an entry corresponding to an SVC code which is associated with an input/output control instruction or a value other than (FF)16 for other entry. The emulation microprogram tests the entry value of the table 42. If the value is other than (FF)16 corresponding to an SVC code, it regards the instruction as an SVC instruction that can be processed on the target machine operating system 23 and carries out the emulation to perform SVC operation as predetermined in accordance with the target machine architecture (steps 34-35). If the entry value is (FF)16, the emulation microprogram regards the instruction as an SVC instruction which is associated with an input/output control macro instruction, saves the PSW indicating the current emulation mode state to the PSW save area 411 in the local execution list 41 depicted in FIG. 4, and at the same time, loads the new ECP

interrupt PSW (412) indicating the entry address of the EPC 21 for translating the input/output control macro instruction of the target machine into the PSW, then releases the emulation mode and transfers control to the ECP 21 (steps 36-39).

CLPV:
a. detecting an interrupt instruction defined in a first program in which an instruction of said target machine is executed;

CLPV:
b. determining whether or not said interrupt instruction is an input/output control macro instruction by use of an operand value of said interrupt instruction; and

CLPV:
c. if said interrupt instruction is determined to be an instruction associated with an input/output control macro instruction in said step b, bypassing the operation system of said target machine by translating said input/output control macro instruction from said first program into an input/output control macro instruction for a second program, in which an instruction of said native machine is executed, under control of a program for controlling said emulation.

CLPV:
d. causing an interrupt in an operating system for said first program is said interrupt instruction is not determined to be an instruction associated with an input/output control macro instruction in said step b.

CLPV:
a. detecting an interrupt instruction existing in a first program in which an instruction of said target machine is executed;

CLPV:
b. searching a table by use of an operand of said interrupt instruction;

CLPV:
e. loading said program status word as an interrupt in an emulation program; and

Full | Title | Citation | Front | Review | Classification | Date | Reference | Claims | KWIC | Draw Desc | Image

Generate Collection

Term	Documents
INTERRUPT$	0
INTERRUPT.USPT.	80738
INTERRUPTA.USPT.	1
INTERRUPTABILITY.USPT.	38
INTERRUPTABLE.USPT.	1063
INTERRUPTABLE-POWER.USPT.	2
INTERRUPTABLE-TYPE.USPT.	1
"INTERRUPTABLE.SUB".USPT.	1
INTERRUPTABLE:.USPT.	2
INTERRUPTABLILITY.USPT.	1
(L1 AND (INTERRUPT$)).USPT.	4

WEST

Generate Collection

Search Results - Record(s) 1 through 1 of 1 returned.

□ 1. Document ID: US 5678032 A

L8: Entry 1 of 1 File: USPT Oct 14, 1997

DOCUMENT-IDENTIFIER: US 5678032 A
TITLE: Method of optimizing the execution of program instuctions by an emulator using a plurality of execution units

ABPL:
An application such as an interpretative emulator executes a wide range of different classes of emulated program instructions developed for the processor architecture being emulated on a host system which includes an dual integer pipelined execution unit. The sets of RISC instructions which execute emulated program instructions are organized within the emulator so as to be processed as two distinct instruction streams by the dual integer pipelined execution units wherein one of the pipelined unit performs the steps necessary to completing a current or foreground like operation on each emulated program instruction while the other pipelined unit performs the steps of an anticipated lookahead or background like operation on the next emulated program instruction. By having one unit execute operations such as interpreting each program instruction within an established foreground instruction stream and the other unit execute operations such as prefetching each next program instruction within an established background instruction stream, the speed of emulated program execution is optimized.

BSPR:
The organization of the present invention focuses on establishing parallel instruction streams of execution at a higher level of an interpreter unit to exploit fully, the capabilities of the host system execution unit architecture which in the preferred embodiment utilizes two integer pipelined execution units. The organization allows one of the dual execution units to perform the steps necessary to completing a current or foreground like operation on each emulated program instruction while the other execution unit performs the steps of an anticipated lookahead or background like operation on each emulated program instruction. By having one functional unit execute operations such as interpreting each emulated program instruction contained in a foreground instruction stream and the other functional unit execute operations such as prefetching each next emulated program instruction contained in a background instruction stream, interpreter unit performance is optimized.

DEPR:
FIGS. 2 & 3--CPU Dual Pipelined Execution Unit Architecture

DEPR:
The interpreter unit 72 of the present invention is organized in a way which exploits to the extent possible, the performance advantages of the above described CPU dual pipelined execution unit architecture in optimizing emulator instruction execution. This is accomplished by organizing the interpreter unit 72 as diagramatically illustrated in FIG. 4. Also, such performance is achieved by including a mechanism within the emulator exception handler 74 which ensures that performance is maintained notwithstanding the occurrence of segment violation types of exception conditions.

DEPR:
FIG. 7 illustrates the pipelined execution of the LDH instruction which is quite similar to the LDR instruction of FIG. 6 in the absence of a segment violation.

The code fragments designated on the left side of FIG. 7 are the same as those indicated in the timing state portion of the figure with different labels showing the transitions between the different code fragments/routines within the interpreter unit 72. As in the previous instruction example, such code fragments are discontiguous sections of code in the interpreter unit 72 as illustrated in Appendix II.

DEPR:
Since this emulated program instruction is performed in a manner similar to the first emulated program instruction example, it will not be discussed in detail. However, it will be noted that in the case of a segment violation, the catcher mechanism is invoked and following the processing of the violation by the mechanism in the manner described above, the interpreter unit continues execution of the RISC instructions as indicated (start of clock cycle 9). It will be appreciated that there is a break in execution after which pipelined operation begins. Because of this, the RISC instructions designated in FIG. 7 may not appear exactly as they would in the absence of the violation. This area of FIG. 7 is shown with a double-line border.

DEPR:
It will be apparent to those skilled in the art that many changes may be made to the preferred embodiment of the present invention. For example, while the present invention is described in terms of an emulator application, the teachings of the present invention can be applied to other applications which could take advantage of using two separate independent instruction streams, a first stream for performing anticipation type operations and a second stream for performing operations utilizing the results of the first stream to complete an execution operation. Also, while the preferred embodiment discloses the use of a specific pipelined execution unit architecture, the teachings of the present invention may be used with other architectures and other types of processors which provide for parallel instruction execution. Also, while the preferred embodiment disclosed the use of two execution units for carrying out the processing of two instruction streams, more such units may be employed.

CLPR:
8. The method of claim 1 wherein n equals two and one of the pipelined integer units executes a foreground RISC instruction stream and another of the pipelined integer units executes a background RISC instruction stream.

Full | Title | Citation | Front | Review | Classification | Date | Reference | Claims | KWIC | Draw Desc | Image

Generate Collection

WEST

Generate Collection

Search Results - Record(s) 1 through 1 of 1 returned.

☐ 1. Document ID: US 5678032 A

```
L9: Entry 1 of 1                    File: USPT                    Oct 14, 1997

DOCUMENT-IDENTIFIER: US 5678032 A
TITLE: Method of optimizing the execution of program instuctions by an emulator
using a plurality of execution units


DEPR:
FIG. 3 illustrates in greater detail, the organization of the FXU 58-4. The FXU
58-4 performs all storage references, integer arithmetic and logical operations.
The FXU 58-4 contains thirty-two 32 bit general purpose registers (GPRs)
implemented by two multiport register files 58-41, two fixed point execution
units 58-43 and 58-44, a data cache directory 58-46 which couples to the control
circuits of block 58-47, and the TLB and Segment Registers 58-48.
```

Full | Title | Citation | Front | Review | Classification | Date | Reference | Claims | KWIC | Draw Desc | Image

Generate Collection

Term	Documents
TLB$	0
TLB.USPT.	1586
TLBA.USPT.	3
TLBABSTEII.USPT.	1
TLBACA.USPT.	2
TLBACACO.USPT.	1
TLBACACU.USPT.	6
TLBACACUO.USPT.	93
"TLBACACUO.SUB.X".USPT.	2
TLBACACU-OXIDE.USPT.	3
(L1 AND (TLB$)).USPT.	1

There are more results than shown above. Click here to view the entire set.

Display 10 Documents, starting with Document: 1

DB Name	**Query**	**Hit Count**	**Set Name**
USPT	(emulator$ or simulator$) same (x86 and risc)	9	L4
USPT	(tapestry adj code$)	0	L3
USPT	(taxi adj translat$)	0	L2
USPT	(taxi adj code$)	0	L1

WEST

□ Generate Collection

L4: Entry 1 of 9 File: USPT Mar 6, 2001

DOCUMENT-IDENTIFIER: US 6199152 B1
TITLE: Translated memory protection apparatus for an advanced microprocessor

BSPR:
However, even though RISC computer systems running emulator software are often capable of running X86 (or other) programs, they usually do so at a rate which is substantially slower than the rate at which state of the art X86 computer systems run the same programs. Moreover, often these emulator programs are not able to run all or a large number of the target programs available.

BSPR:
In FIG. 1(b), a typical RISC microprocessor such as a PowerPC microprocessor used in an Apple Macintosh computer is represented running the same target application program which is designed to be run on the CISC processor of FIG. 1(a). As may be seen, the target application is run on the host processor using at least a partial target operating system to respond to a portion of the calls which the target application generates. Typically these are calls to the application-like portions of the target operating system used to provide graphical interfaces on the display and short utility programs which are generally application-like. The target application and these portions of the target operating system are changed by a software emulator such as Soft PC.RTM. which breaks the instructions furnished by the target application program and the application-like target operating system programs into instructions which the host processor and its host operating system are capable of executing. The host operating system provides the interfaces through which access to the memory and input/output hardware of the RISC computer may be gained.

BSPR:
However, the host RISC processor and the hardware devices associated with it in a host RISC computer are usually quite different than are the devices associated with the processor for which the target application was designed; and the various instructions provided by the target application program are designed to cooperate with the device drivers of the target operating system in accessing the various portions of the target computer. Consequently, the emulation program, which changes the instructions of the target application program to primitive host instructions which the host operating system is capable of utilizing, must somehow link the operations designed to operate hardware devices in the target computer to operations which hardware devices of the host system are capable of implementing. Often this requires the emulator software to create virtual devices which respond to the instructions of the target application to carry out operations which the host system is incapable of carrying out because the target devices are not those of the host computer. Sometimes the emulator is required to create links from these virtual devices through the host operating system to host hardware devices which are present but are addressed in a different manner by the host operating system.

BSPR:
Target programs when executed in this manner run relatively slowly for a number of reasons. First, each target instruction from a target application program and from the target operating system must be changed by the emulator into the host primitive functions used by the host processor. If the target application is designed for a CISC machine such as an X86, the target instructions are of varying lengths and quite complicated so that changing them to host primitive instructions is quite involved. The original target instructions are first decoded, and the sequence of primitive host instructions which make up the target instructions are determined. Then the address (or addresses) of each sequence of primitive host instructions is determined, each sequence of the primitive host instructions is fetched, and these primitive host instructions are executed in or out of order. The large number of extra steps required by an emulator to change

the target application and operating system instructions into host instructions understood by the host processor must be conducted each time an instruction is executed and slows the process of emulation.

BSPR:
In FIG. 1(c), another example of emulation is shown. In this case, a PowerPC microprocessor used in an Apple Macintosh computer is represented running a target application program which was designed to be run on the Motorola 68000 family CISC processors used in the original Macintosh computers; this type of arrangement has been required in order to allow Apple legacy programs to run on the Macintosh computers with RISC processors. As may be seen, the target application is run on the host processor using at least a partial target operating system to respond to the application-like portions of the target operating system. A software emulator breaks the instructions furnished by the target application program and the application-like target operating system programs into instructions which the host processor and its host operating system are capable of executing. The host operating system provides the interfaces through which access to the memory and input/output hardware of the host computer may be gained.

BSPR:
Again, the host RISC processor and the devices associated with it in the host RISC computer are quite different than are the devices associated with the Motorola CISC processor; and the various target instructions are designed to cooperate with the target CISC operating system in accessing the various portions of the target computer. Consequently, the emulation program must link the operations designed to operate hardware devices in the target computer to operations which hardware devices of the host system are capable of implementing. This requires the emulator to create software virtual devices which respond to the instructions of the target application and to create links from these virtual devices through the host operating system to host hardware devices which are present but are addressed in a different manner by the host operating system.

BSPR:
In FIG. 1(d), a particular method of emulating a target application program on a host processor which provides relatively good performance for a very limited series of target applications is illustrated. The target application furnishes instructions to an emulator which changes those instructions into instructions for the host processor and the host operating system. The host processor is a Digital Equipment Corporation Alpha RISC processor, and the host operating system is Microsoft NT. The only target applications which may be run by this system are 32 bit applications designed to be executed by a target X86 processor with a Windows WIN32s compliant operating system. Since the host and target operating systems are almost identical, being designed to handle these same instructions, the emulator software may change the instructions very easily. Moreover, the host operating system is already designed to respond to the same calls that the target application generates so that the generation of virtual devices is considerably reduced.

BSPR:
Although this is technically an emulation system running a target application on a host processor, it is a very special case. Here the emulation software is running on a host operating system already designed to run similar applications. This allows the calls from the target applications to be more simply directed to the correct facilities of the host and the host operating system. More importantly, this system will run only 32 bit Windows applications which probably amount to less than one percent of all X86 applications. Moreover, this system will run applications on only one operating system, Windows NT; while X86 processors run applications designed for a large number of operating systems. Such a system, therefore, could be considered not to be compatible within the terms expressed earlier in this specification. Thus, a processor running such an emulator cannot be considered to be a competitive X86 processor.

DEPR:
During the following description, in some cases the target program is referred to as a program which is designed to be executed on an X86 microprocessor in order to provide exemplary details of operation because the majority of emulators run X86 applications. However, the target program may be one designed to run on any family of target computers. This includes target virtual computers, such as Pcode

machines, Postscript machines, or Java virtual machines.

WEST

☐ Generate Collection

L4: Entry 2 of 9 File: USPT Feb 29, 2000

DOCUMENT-IDENTIFIER: US 6031992 A
TITLE: Combining hardware and software to provide an improved microprocessor

BSPR:
However, even though RISC computer systems running emulator software are often capable of running X86 (or other) programs, they usually do so at a rate which is substantially slower than the rate at which state of the art X86 computer systems run the same programs. Moreover, often these emulator programs are not able to run all or a large number of the target programs available.

BSPR:
In FIG. 1(b), a typical RISC microprocessor such as a PowerPC microprocessor used in an Apple Macintosh computer is represented running the same target application program which is designed to be run on the CISC processor of FIG. 1(a). As may be seen, the target application is run on the host processor using at least a partial target operating system to respond to a portion of the calls which the target application generates. Typically these are calls to the application-like portions of the target operating system used to provide graphical interfaces on the display and short utility programs which are generally application-like. The target application and these portions of the target operating system are changed by a software emulator such as Soft PC.RTM. which breaks the instructions furnished by the target application program and the application-like target operating system programs into instructions which the host processor and its host operating system are capable of executing. The host operating system provides the interfaces through which access to the memory and input/output hardware of the RISC computer may be gained.

BSPR:
However, the host RISC processor and the hardware devices associated with it in a host RISC computer are usually quite different than are the devices associated with the processor for which the target application was designed; and the various instructions provided by the target application program are designed to cooperate with the device drivers of the target operating system in accessing the various portions of the target computer. Consequently, the emulation program, which changes the instructions of the target application program to primitive host instructions which the host operating system is capable of utilizing, must somehow link the operations designed to operate hardware devices in the target computer to operations which hardware devices of the host system are capable of implementing. Often this requires the emulator software to create virtual devices which respond to the instructions of the target application to carry out operations which the host system is incapable of carrying out because the target devices are not those of the host computer. Sometimes the emulator is required to create links from these virtual devices through the host operating system to host hardware devices which are present but are addressed in a different manner by the host operating system.

BSPR:
Target programs when executed in this manner run relatively slowly for a number of reasons. First, each target instruction from a target application program and from the target operating system must be changed by the emulator into the host primitive functions used by the host processor. If the target application is designed for a CISC machine such as an X86, the target instructions are of varying lengths and quite complicated so that changing them to host primitive instructions is quite involved. The original target instructions are first decoded, and the sequence of primitive host instructions which make up the target instructions are determined. Then the address (or addresses) of each sequence of primitive host instructions is determined, each sequence of the primitive host instructions is fetched, and these primitive host instructions are executed in or out of order. The large number of extra steps required by an emulator to change

the target application and operating system instructions into host instructions understood by the host processor must be conducted each time an instruction is executed and slows the process of emulation.

BSPR:
In FIG. 1(c), another example of emulation is shown. In this case, a PowerPC microprocessor used in an Apple Macintosh computer is represented running a target application program which was designed to be run on the Motorola 68000 family CISC processors used in the original Macintosh computers; this type of arrangement has been required in order to allow Apple legacy programs to run on the Macintosh computers with RISC processors. As may be seen, the target application is run on the host processor using at least a partial target operating system to respond to the application-like portions of the target operating system. A software emulator breaks the instructions furnished by the target application program and the application-like target operating system programs into instructions which the host processor and its host operating system are capable of executing. The host operating system provides the interfaces through which access to the memory and input/output hardware of the host computer may be gained.

BSPR:
Again, the host RISC processor and the devices associated with it in the host RISC computer are quite different than are the devices associated with the Motorola CISC processor; and the various target instructions are designed to cooperate with the target CISC operating system in accessing the various portions of the target computer. Consequently, the emulation program must link the operations designed to operate hardware devices in the target computer to operations which hardware devices of the host system are capable of implementing. This requires the emulator to create software virtual devices which respond to the instructions of the target application and to create links from these virtual devices through the host operating system to host hardware devices which are present but are addressed in a different manner by the host operating system.

BSPR:
In FIG. 1(d), a particular method of emulating a target application program on a host processor which provides relatively good performance for a very limited series of target applications is illustrated. The target application furnishes instructions to an emulator which changes those instructions into instructions for the host processor and the host operating system. The host processor is a Digital Equipment Corporation Alpha RISC processor, and the host operating system is Microsoft NT. The only target applications which may be run by this system are 32 bit applications designed to be executed by a target X86 processor with a Windows WIN32s compliant operating system. Since the host and target operating systems are almost identical, being designed to handle these same instructions, the emulator software may change the instructions very easily. Moreover, the host operating system is already designed to respond to the same calls that the target application generates so that the generation of virtual devices is considerably reduced.

BSPR:
Although this is technically an emulation system running a target application on a host processor, it is a very special case. Here the emulation software is running on a host operating system already designed to run similar applications. This allows the calls from the target applications to be more simply directed to the correct facilities of the host and the host operating system. More importantly, this system will run only 32 bit Windows applications which probably amount to less than one percent of all X86 applications. Moreover, this system will run applications on only one operating system, Windows NT; while X86 processors run applications designed for a large number of operating systems. Such a system, therefore, could be considered not to be compatible within the terms expressed earlier in this specification. Thus, a processor running such an emulator cannot be considered to be a competitive X86 processor.

DEPR:
During the following description, in some cases the target program is referred to as a program which is designed to be executed on an X86 microprocessor in order to provide exemplary details of operation because the majority of emulators run X86 applications. However, the target program may be one designed to run on any family of target computers. This includes target virtual computers, such as Pcode

WEST

□ Generate Collection

L4: Entry 7 of 9 File: USPT Jul 14, 1998

DOCUMENT-IDENTIFIER: US 5781750 A
TITLE: Dual-instruction-set architecture CPU with hidden software emulation mode

BSPR:
Because of the huge amount of software written for the x86 instruction set, software programs called "emulators" have been written that allow x86 code to be run on RISC CPU's. The emulator must parse the x86 code and convert the x86 instructions into RISC code. This can be done before the x86 code is executed, which is known as binary translation because the x86 code is translated at a low or binary level and translated to the RISC instruction set. This is an undesirable method since binary translation requires 5 to 10 times as long to run as a native x86 program. A native program is one that is written in the lowest-level machine instructions for that architecture.

BSPR:
Since there is so much installed x86 code, it is greatly desired to run x86 programs on newer RISC CPU's, but without the performance degradation of the software emulator. One approach would be to add support in hardware for all the x86 instructions. While this would provide the highest performance, the complexity would be enormous, and the cost very high. A very large microcode, composed of small sub-instructions or micro-instructions, could be constructed to allow execution of all x86 instructions. This would require a large microcode read-only memory (ROM) on the CPU die. Various subroutines in the microcode would control execution of the different instructions. While it would be a tremendous competitive advantage to be able to run native x86 code on a RISC CPU, no manufacturer has yet been able to achieve this, attesting to the great technical difficulty of integrating the entire x86 instruction set.

DEPR:
When trying to implement both a RISC and a CISC architecture on the same CPU, a software emulator suffers from poor performance, while a hardware-only approach is too complex and expensive. An approach that uses some hardware and some software is the best, because simple, fast instructions can be implemented by hardware on the CPU, while complex instructions can be detected by the hardware and trapped to a software emulation driver. Thus some of the complexity of the CISC architecture is moved to the software driver. However, the emulation driver must be isolated and hidden from the user's code being executed, otherwise the user programs could modify or destroy the emulation code, resulting in a system crash.

DEPR:
The RISC sub-block of instruction decode 36 decodes the PowerPC RISC instruction set. All instructions are 32 bits in size, and some require two levels of instruction decoding. The first level determines the basic type of instruction and is encoded in the 6 most significant bits. Table 2 shows the 64 possible basic or primary opcode types. For example, 001110 binary (0E hex) is ADDI - add with an immediate operand, while 100100 (24 hex) is STW - store word. The CPU executes the 45 unshaded opcodes directly in hardware. The fifteen darkly shaded opcodes, such as 000000, are currently undefined by the PowerPC architecture. Undefined opcodes force the CPU into emulation mode, where the emulation driver executes the appropriate error routine. Should instructions later be defined for these opcodes, an emulator routine to support the functionality of the instruction could be written and added to the emulator code. Thus the CPU may be upgraded to support future enhancements to the PowerPC instruction set. It is possible that the CPU could be field-upgradable by copying into emulation memory a diskette having the new emulation routine.

WEST

□ Generate Collection

L4: Entry 8 of 9 File: USPT Jul 14, 1998

DOCUMENT-IDENTIFIER: US 5781457 A
TITLE: Merge/mask, rotate/shift, and boolean operations from two instruction sets executed in a vectored mux on a dual-ALU

BSPR:
It is greatly desired to execute both RISC and x86 CISC instructions on the same central processing unit (CPU). This could allow a RISC computer to execute newer PowerPC.TM. RISC programs and also execute older x86 CISC programs. A vast amount of code has been written with native x86 instructions which currently can only be emulated in software on RISC computers. Software emulation is slow and thus little or no performance gain is observed when running the CISC programs on emulators on RISC computers.

WEST

□ Generate Collection

L4: Entry 9 of 9 File: USPT Nov 12, 1996

DOCUMENT-IDENTIFIER: US 5574927 A
TITLE: RISC architecture computer configured for emulation of the instruction set of a target computer

ORPL:
Selzer; "Four ways to Fake an X86; Picks Strengths, Minuses to Determine Suitability for Tasks; Software Emulators rarely replace an X86 system; Cloners see Their Future in RISC Cores". PC Week; Nov. 1994.

US 094290940CP1

Creation date: 10-14-2004
Indexing Officer: OAWONIYI - OLUSINMISLOA AWONIYI
Team: OIPEBackFileIndexing
Dossier: 09429094

Legal Date: 04-10-2001

No.	Doccode	Number of pages
1	CTNF	7
2	1449	1

Total number of pages: 8

Remarks:

Order of re-scan issued on ....